The Textbook

THIRD EDITION

Syed Mansoor Sarwar
Robert M. Koretsky

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

UNIX
The Textbook

THIRD EDITION

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

UNIX
The Textbook

THIRD EDITION

Syed Mansoor Sarwar
Robert M. Koretsky

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

© 2017 by Taylor & Francis Group, LLC
CRC Press is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works

Printed on acid-free paper
Version Date: 20160226

International Standard Book Number-13: 978-1-4822-3358-2 (Hardback)

This book contains information obtained from authentic and highly regarded sources. Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, please access www.copyright.com (http://www.copyright.com/) or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

Library of Congress Cataloging-in-Publication Data

Names: Sarwar, Syed Mansoor, author. | Koretsky, Robert, author.
Title: UNIX : the textbook / Syed Mansoor Sarwar and Robert M. Koretsky.
Description: Third edition. | Boca Raton : Taylor & Francis, CRC Press, 2016.
| Includes bibliographical references and index.
Identifiers: LCCN 2016009010 | ISBN 9781482233582 (alk. paper)
Subjects: LCSH: UNIX (Computer file) | Operating systems (Computers)
Classification: LCC QA76.774.U64 S37 2016 | DDC 005.4/32--dc23
LC record available at http://lccn.loc.gov/2016009010

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

To my family

S.M.S.

To my family

R.M.K.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Contents

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Preface to the Third Edition

This third edition of *UNIX: The Textbook* has many significant changes and additions incorporated into it, in terms of both the scope and content of the previous editions. It is a textbook on the modern, twenty-first-century UNIX operating system. It uses an introductory approach in style, very similar to the style of the previous editions. With the exception of four chapters on system programming, the book can be used very successfully by a complete novice, as well as by an experienced UNIX system user, in both an informal and formal learning environment.

The two UNIX systems that we deploy to illustrate everything in this edition are PC-BSD and Solaris. There are many things that make these two systems superior to, as well as very different from, any contemporary, nominally UNIX distribution, and also from other NIX-like operating systems, such as Linux and OS X. There are many topics covered in this book that older, more traditional textbook approaches to UNIX could not include, such as the Zettabyte File System (ZFS) and a highly developed KDE or Gnome GUI desktop environment. The traditional text-based command line interface, though, is still a very integral part of our presentation of UNIX.

CHANGES IN THE THIRD EDITION OF THIS BOOK

Because PC-BSD and Solaris UNIX have had many important functional additions made to the application user interface since the previous edition came out, and because UNIX is now an even more widely-dispersed system in the marketplace than previously, we felt that we needed to add instructional material to the book covering these additions, including:

- Showing desktop KDE PC-BSD and Gnome Solaris as base system implementations of UNIX.
- Adding methods for customizing vi, vim, and emacs.
- Adding a complete tutorial chapter on the Python programming language and its use in UNIX.
- Giving a complete tutorial on the `git` command, and using Github.
- Adding four new, complete chapters on UNIX system programming and the UNIX API.

- Revising the chapter on networking and internetworking to bring it in line with current standards.
- Complete covering system call interfaces, files, file-related data structures in the UNIX kernel, file I/O paradigms, and file manipulation API.
- Extensive coverage of UNIX processes and threads, process-related kernel data structures in the UNIX kernel, process management API, and signal handling.
- Comprehensively covering interprocess communication in UNIX using pipes, named pipes (FIFOs), and sockets.
- Comprehensively coverage of Internetworking with UNIX TCP/IP: the client–server software for the Internet services using sockets, including the design and implementation of concurrent servers using the `select` system call and the need for concurrent clients.
- Providing coverage of important practical considerations in the design and implementation of production-quality client–server software.
- Completely revising much of the tutorial section on the X Window System to now include writing Xlib and Xcb code.
- Adding a new, extensive chapter on UNIX system administration that details installation, maintenance, and updating/upgrading PC-BSD and Solaris systems on your own PC.
- Adding a complete reference chapter on ZFS, the default file system on PC-BSD and Solaris.
- Adding a complete chapter on virtualization methodologies that illustrate PC-BSD jails and iocage, Solaris zones, and installation of various guest operating systems in popular host systems using VirtualBox.
- Adding many new diagrams, tables, interactive shell sessions, in-chapter tutorials, in-chapter exercises, and end-of-chapter problems.
- Providing coverage of many new commands and enhancing coverage of existing commands.
- Providing up-to-date URLs for important Web resources on nearly everything in the book.
- Enhancing the usability of all shell scripts, Python and C programs, and other programming code shown in the printed book, by installing them at a Github repository for easy download to a local repository.
- Redesigning the text layout to provide a more usable active learner document.

As in the last editions, one very important fact to keep in mind when you look at what we have included in this edition, and for that matter in the sequencing and presentation of all the material in the book, is the fact that we have almost 65 years of practical teaching experience at the college level. Our continuing concept for this book grew out of our unwillingness to use either the large, intractable UNIX reference sources or the short "nutshell" guides to teach meaningful, complete, and relevant introductory classes on the subject. We still feel very strongly that a textbook approach, with pedagogy incorporating in-chapter tutorials and exercises, as well as useful problem sets at the end of each chapter, allows us to present all the important UNIX topics for a classroom lecture–laboratory–homework presentation. We have continued to fine-tune this textbook presentation in a manner consistent with optimal learning outcomes (i.e., well-thought-out sequencing of old and new topics, well-developed and timely lessons, online laboratory problems, and homework exercises/problems synchronized with the sequencing of chapters in the book). As in the earlier editions, because of the greatly increased depth and breadth of coverage of the basic and advanced topics we present, anyone interested in furthering their professional knowledge of the subject matter will also find this textbook useful.

THE PURPOSES OF THIS BOOK IN THE THIRD EDITION

Our primary purpose remains a didactic description of the UNIX application user's interface (AUI), and we also try to do this in a way that gives the reader insight into the inner workings of the system, along with explaining some important UNIX concepts, data structures, and algorithms. Notable examples of our revealing the inner workings of the system are the in-depth descriptions of the UNIX file, process, and I/O redirection concepts.

Our secondary purpose is to extensively describe the UNIX application programmer's interface (API) in terms of C/C++ libraries and UNIX system calls. In writing this third edition, particularly for the system programming chapters, we do assume previous basic to intermediate knowledge of C/C++ programming on the part of the reader.

The tertiary purpose of this textbook is to describe some important UNIX software engineering tools for developers of C/C++ software and shell scripts.

THE PRESENTATION FORMAT

The didactic structure of each chapter in this new edition follows one of two similar formats: either the shell session format, or the tutorial format. In the shell session format (used in all chapters except 3, 16, 22–25), the following outline is used:

- Learning objectives
- Introduction
- Topic discussion and background organized in sections and subsections
- Illustrative commands or topic illustrations presented as *shell sessions*, where the user types in commands shown and results are displayed
- In-chapter exercises that reinforce what was discussed on a topic or done interactively in a shell session

- Summary
- End-of-chapter questions and problems keyed to topics presented

In the tutorial format (used in Chapters 3, 16, 22–25) the following outline is used:

- Learning objectives
- Introduction
- Topics discussions and background organized in sections and subsections
- One or several example sessions or practice session tutorials that illustrate the commands and topics of interest in any particular section or subsection
- Illustrative commands or topic illustrations presented as shell sessions, where the user types in commands shown and results are displayed
- In-chapter exercises that reinforce what was discussed in an example or practice session, on a topic, or done interactively in a shell session
- Summary
- End-of-chapter problems keyed to topics presented

This edition has had many new diagrams and tables added, and there are many new in-chapter tutorials, interactive shell sessions, in-chapter exercises, and end-of-chapter problems. We have added more syntax boxes whenever we introduce a new command or utility. These syntax boxes describe the exact syntax of the command (and any other pertinent variants of the basic syntax), its purpose, the output produced by the command, and its useful options and features. In addition, every chapter contains a summary of the material covered in the chapter. There is also a glossary of terms used in the book.

PATHWAYS THROUGH THE TEXT

If this book is to be used as the main text for an introductory course in UNIX, Chapters 1–15 and 22 should be covered. If the book is to be used as a companion to the main text in an operating systems concepts and principles course, the coverage of chapters would be dictated by the order in which the main topics of the course are covered but should include Chapters 4, 9, 10, 18–21, 24, and 25. For use in a C/C++ or shell programming course, Chapters 1, 4–17 and relevant sections of Chapter 3 would be a great help to students. The extent of coverage of Chapter 17 would depend on the nature of the course—partial coverage in an introductory and full coverage in an advanced course. For use in a course of UNIX system administration course, Chapters 4–11, 22, 23, 24, 25, and relevant sections of Chapters 12–15 should be used. In a system programming course, Chapters 12–16, 18–21, and relevant portions of Chapters 4–11 and Chapters 24–25 should be covered. Finally, in a course on UNIX network programming, Chapters 11, 18–21, and relevant portions of Chapters 12–16 should be used.

THE DESIGN OF FONTS

The following typefaces have been used in the book for various types of text items.

Font	Text Item
Minion Pro, italic	Key terms:
	• Whatever directory you are currently in is known as the *present working directory*.
Minion Pro, bold	Files/directories/symbolic constants/menu paths:
	• The directory **first** and the file **myfile2** are now removed.
	• Make the Options menu choice **Save Options**
	• Make the pull-down menu choice **File>Quit**
	• A socket with **AF_INET** address family is known as the *Internet domain socket*.
`Courier Std`	Commands, program code, output of commands and programs, and options:
	• Use the `man` and `whatis` commands to find information about the `passwd` command.
	• The output of the `date` command is `Thu Apr 7 13:53:30 PKT 2016.`
	• You can use the `–l` option to display the long listing.
	• The following session shows the Bourne shell script in the **for_demo1** file
	`$` **`cat for_demo1`**
	`#!/bin/sh`
	`for people in Debbie Jamie John Kitty Kuhn Shah`
	`do`
	`        echo "people"`
	`done`
	`exit 0`
	`$`
	Keystrokes:
	• `<Enter>`, `<Alt+V>`, `<F1>`, `a`
	Prompts, messages, dialogs, windows:
	• A user who runs a `write` or `talk` command sees the message `Permission denied`.
	• The system then displays the `login:` prompt.
	• In the `Find file:` dialog box that opens…
	• Click the OK button in the `Save` window.
`Courier Std, bold`	User input:
	• `[bob@pcbsd-923] ~%` **`ssh 192.168.0.8`**
	• `Password for bob@pcbsd-2467:` **`XXX`**

SUPPLEMENTS

A variety of supplemental materials are available for all users of this textbook and additional material only to qualified instructors.

Materials Available to all Users of this Textbook

1. You can use your web browser and retrieve the materials from the following Github repository:

   https://github.com/bobk48/unixthetextbook3

2. Or you can use the steps of the following example in the book to prepare and download these materials:

   Example 17.5: Pulling from a GitHub Repository

3. To access these materials as shown in Example 17.5, pull from the repository using this **`git`** command:

   ```
   % git pull https://github.com/bobk48/unixthetextbook3 master
   ```

4. In either case, you will find the following in your new local repository:

- Answers to in-chapter exercises.
- Source code for C/C++ programs, Python code, and long shell scripts, arranged by chapter.
- Updated links to other UNIX resources on the Web.
    - Author-maintained Web Resources listing for chapters which do not contain this in the printed book.
    - Updates of version-specific content of PC-BSD and Solaris that severely impact our printed-book presentations.
    - Errata.

  Additional material is available from the CRC website at http://www.crcpress.com/product/isbn/97814822335832.

Resources Available to Qualified Instructors Only

Contact your CRC Press representative to gain access to this material.

Solutions to problems at the end of each chapter.

We take full responsibility for any errors in the book. You can send your error reports and comments to us at the above listed Github site. We will incorporate your feedback and fix any errors in subsequent printings.

Acknowledgments for the Third Edition

We started writing this third edition during the fall of 2013 and finished it in the spring of 2016. Completing such a large revision would not have been possible without the help of many. First and foremost, the authors sincerely thank the editor of the book, Randi Cohen, for her support, guidance, reassurance, professionalism, and understanding throughout this project. She is the top of the line. Also, we would like to extend a warm and gracious acknowledgment to Amber Donley and Joette Lynch at CRC Press and Michelle van Kampen at Deanta Publishing Services for all of their professional diligence and extremely thorough work on the production of this book.

We convey our sincere thanks to the following reviewers of this edition who gave valuable feedback, and numerous accurate and insightful comments. We are particularly grateful to Professor Richard Fox, who meticulously read the manuscript and gave many useful suggestions that greatly enhanced the final product.

- Hussein Abdel-Wahab   Old Dominion University
- Gregory B. Newby   Compute Canada
- Richard Fox   Northern Kentucky University

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Acknowledgments for the Second and First Editions

Thanks are extended to Marilyn Lloyd of AWL for her many excellent suggestions and outstanding work in the production phase of this book. We are also grateful to Joyce Wells for designing a beautiful cover for the book. Thanks also to Katherine Harutunian at AWL for her many valuable suggestions and support. The work of the whole Addison Wesley team was excellent! Special thanks to Keith Henry and Alisa Andreola at Dartmouth Publishing for their excellent copyediting and interior design.

We also thank the following reviewers for reviewing the second edition of the book.

• Robert Albright	University of Delaware
• Hussein Abdel-Wahab	Old Dominion University
• Dunren Che	Southern Illinois University
• C. Michael Costanzo	University of California, Santa Barbara
• Robert M. Cubert	University of Florida
• James P. Durbano	University of Delaware
• Nisar Hundewale	Georgia State University
• Mark Hutchenreuther	California Polytechnic State University
• Stephen P. Leach	Florida State University
• Susan Lincke-Salecker	University of Wisconsin Parkside
• Mike Qualls	Grossmont Community College
• Daniel Tomasevich	San Francisco State University
• Paul Tymann	Rochester Institute of Technology
• Troy Vasiga	University of Waterloo

We would like to acknowledge Ronald E. Bass, Thomas A. Burns, Chuck Lesko, Toshimi Minoura, Selmer Moen, Gregory B. Newby, Dr. Marianne Vakalis, and Dr. G. Jan Wilms for their insightful comments in reviewing this book's first edition.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Personal Acknowledgments

Syed Mansoor Sarwar I thank my parents, wife, children, and siblings for their love, support, and trust. They have all been a positive influence in my life, have helped me in many ways, and remain my biggest supporters. I can never pay back the grace that they have extended to me over the years. My mother, a class act in motherhood, has been fighting serious illnesses for the past several years. I pray for her good health and peace of mind. My father, an avid reader even at the age of 94, is an icon of wisdom, courage, care, intellect, logical reasoning, and mental toughness, and remains my inspiration for the quest of knowledge discovery, rational thinking, and service to fellow humans. Without his and my wife's continuous encouragement and support, writing this book would not have been possible.

My wife, Robina, and children Maham, Ibraheem, and Hassan have been extremely patient and supportive during the course of this project. Thank you for your understanding and support, guys! I could not have done it without you. At the time of writing the second edition of the book, Hassan and Maham were in middle school, and the then "snug bug" Ibraheem in prenursery. Now, Hassan is the CEO of *infinione* (infinione.com), a Los Angeles-based technology company that he founded when he was a junior at the University of Southern California (USC). Maham is a merit scholarship holder senior, majoring in textile design with the Best Designer award under her belt in a national competition. Several of her designs have already been marketed through well-known fashion women's wear brands such as Beechtree. Ibraheem is now a young 6′2″ teenager, with a love for mathematics and science, and has the goal to become a top-notch scientist as well as a professional basketball player.

My special thanks to my sisters Rizwana and Farhana, and brothers Aqeel, Nadeem, Masood, and Nabeel for their friendship and care. I convey my sincere gratitude to them, my sisters-in-law Maimoona, Sadaf, and Farzana, and brother-in-law Hamid, for taking care of our father and ailing mother during the hours of their need. Folks, I will forever remain indebted to you for performing my share of service toward our parents. I hope to be able to offer a payback in some form someday.

I thank the teachers who taught me the use, administration, internals, and programming of UNIX. They are James Davis, Doug Jacobson, and Arthur Oldehoeft at Iowa State University, and Jim Binkley at Oregon Graduate Institue and Portland State University. I wrote this book under the most trying professional circumstances of my career. I thank my colleagues at the Punjab University College of Information Technology (PUCIT), the

GIS Center, the Institute of Business and Information Technology (IBIT), Institute of Chemistry, and the Department of Space Science who supported me through thick and thin. I wish them all peaceful lives and outstanding professional careers. Fellows, you are the best colleagues one could ever wish for. Thank you for being there!

I also thank my friends, colleagues, teachers, students, and professionals who have played a positive role in my life. They are in no particular order: R. A. Khan, Mohammad Ishaq, the late Ghulam us Saqlain Naqvi, Shahid H. Bokhari, Mohammad Ali Maud, Haroon Babri, Wasif Khan, Norman Scott, Janice Jenkins, Art Pohm, James Davis, Arthur Oldehoeft, Thomas Piatkowski, Douglas Jacobson, the late James "Jolly" Triska, the late Charlie Wright, David Schmidt, Terry Smay, Zartash Uzmi, Tariq Jadoon, Sohaib Khan, Aamer Mahmood, Asim Loan, the late Masood Ahmad, Sohail Aftab, Jahangir Ikram, Tariq Butt, Asim Rasul, Hamid Ch, Ejaz Ashraf, Shahzad Sarwar, Murtaza Yousaf, Waqar Jaffrey, Kamran Malik, Sadeeqa Khan, Shahid Farid, Hassan Khan, Sidra Faisal, Zobia Sohail, Laeeq Aslam, Faisal Bukhari, Waheed Iqbal, Faisal Aslam, Fawaz Bokhari, Nastaeen Fatima, Muddassira Arshad, Esha Aftab, Fakhra Jabeen, Aisha Khan, Amina Mustanser, Tayyaba Tariq, Saima Ali, Umair Babar, Imran Farid, Mehvish Poshni, Khurram Shahzad, Zubair Nawaz, Adeel Nisar, Ahmad Ghazali, Asif Sohail, Muhammad Farooq, Kashif Murtaza, Nazar Khan, Aasim Ali, Qudsia Hamid, Muhammad Haris, Nida Samad, Zia Afzal, Syed Muhammad Ali, Samreen Jawed, Anzar Ahmad, Rukhsana Kauser, Madiha Khalid, Mian Mubasher, Sanam Ahmad, Sarah Riaz, Asim Siddique, Tariq Saeed, Mehwish Khurshid, Adnan Asif, Saleem Akhtar, Aamir Raza, Mohsin Omer, Saeed Abbasi, Muhammad Shafiq, Arif Kareem, Muhammad Ajmal, Athar Pasha, Anisul Haq, Shamaila Gull, Javed Sami, Ahsan Sharif, Shaukat Ali, Aurangzeb Alamgir, the late Saeed Nagra, Shahid Ghazi, Asghar Iqbal, Fauzia Adeeba, Maryam Nawaz, Khawaja Imran Nazir, Shahzad Shaukat, Syed Mazahir Ali Akbar Naqvi, Saad Rasool, Anum Azhar, Aitzaz Chaudhary, Aziz Inan, Zia Yamayee, Shahid Ghazi, Asghar Iqbal, Fauzia Adeeba, Maryam Nawaz, Kitty Tilton, Jamie Strohecker, Debbie Spear, Lorraine Yoder, Mathew Kuhn, Tanya Crenshaw, Peter Osterberg, Robert Albright and, last but not least, my mentor at the University of Portland, the late Tom Nelson.

I thank the many students at the Iowa State University, University of Portland, Portland Community College, LUMS, and PUCIT who helped me in the development of the material for the book. Thanks also to my colleagues at the University of Portland, LUMS, and PUCIT who helped me with the installation and use of the UNIX systems for the three editions of the book. They are Kent Thompson, Dale Frakes, Abid Latif "Pasoorri", Arif Butt, and Rizwan Malik. My office staff Zubair Siddique, Muhammad Azhar, and Humayun Ejaz helped me with all the diagrams in this edition of the book. Thank you, gentlemen!

Last but not least, I also thank my coauthor, Bob Koretsky, for his help, encouragement, and trust. Bob, you were a great support throughout the project. Completing this task was not possible without you! I look forward to working with you on future projects, including the second edition of *Linux: The Textbook*.

Robert M. Koretsky I thank my family, for all of the love and continued support through the years. Special thanks to my grandsons Victor and Garvey. I also thank Mansoor Sarwar, for his friendship, intelligence, patience, and continued support and inspiration.

CHAPTER 1

Overview of Operating Systems

Objectives

- To explain what an operating system is
- To describe briefly operating system services
- To describe character and graphical user interfaces
- To discuss different types of operating systems
- To describe briefly the UNIX operating system
- To give an overview of the structure of a contemporary system
- To describe briefly the structure of the UNIX operating system
- To detail some important system setups
- To describe briefly the history of the UNIX operating system
- To provide an overview of the different types of UNIX systems

1.1 INTRODUCTION

Many operating systems are available today, some general enough to run on any type of computer (from a personal computer, or PC, to a mainframe), and some specifically designed to run on a particular type of computer system, including real-time computer systems used to control the movement of mechanical devices such as robots, tablet computers, and cell phones. In this chapter, we describe the purpose of an operating system and the different classes of operating systems. Before describing different types of operating systems and where UNIX fits in this categorization, we present a layered diagram of a contemporary computer system and discuss the basic purpose of an operating system.

We then describe different types of operating systems and the parameters used to classify them. Then, we identify the class that UNIX belongs to and briefly discuss the different members of the UNIX family.

The people who use UNIX comprise application developers, systems analysts, programmers, administrators, business managers, academicians, and people who just wish to read their e-mail. From its earliest inception in 1969 as a laboratory research tool, it was further developed in the academic community, and then endorsed for commercial uses. In its version today, UNIX has an underlying functionality that is complex but easy to learn, and extensible yet easily customized to suit a user's style of computing. One key to understanding its longevity and its heterogeneous appeal is to study the history of its evolution.

1.2 WHAT IS AN OPERATING SYSTEM?

A computer system consists of various hardware and software resources, as shown in a layered fashion in Figure 1.1. The primary purpose of an operating system is to facilitate easy, efficient, fair, orderly, and secure use of these resources. It allows the users to employ application software—spreadsheets, word processors, Web browsers, e-mail software, and other programs. Programmers use language libraries, system calls, and program generation tools (e.g., text editors, compilers, and version control systems) to develop software. Fairness is obviously not an issue if only one user at a time is allowed to use the computer system, including single-user desktop systems, laptops, tablet computers, and cell phones. However, if multiple users are allowed to use the computer system, fairness and security are two main issues to be tackled by the operating system designers.

Hardware resources include keyboards, touch pads, display screens (may also be touch screens), main memory (commonly known as *random access memory* or RAM), disk drives, network interface cards (NICs), and central processing units (CPUs). Software resources include applications such as word processors, spreadsheets, games, graphing

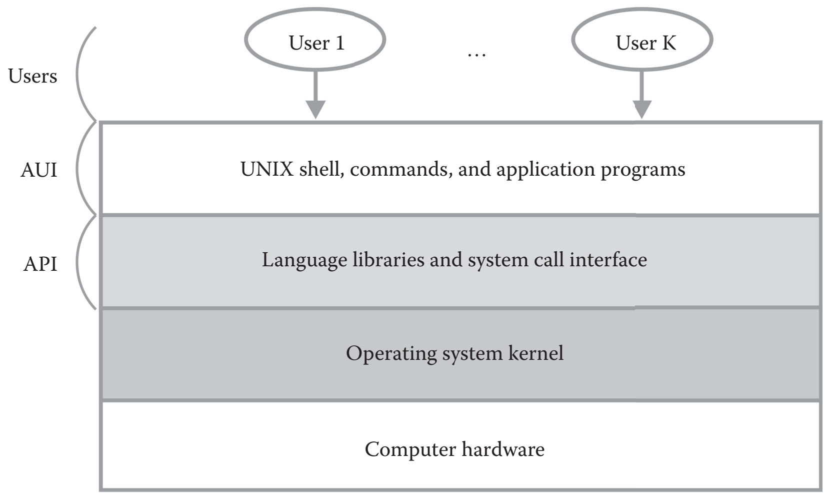

User 1
...
User K
Users
AUI
API
UNIX shell, commands, and application programs
Language libraries and system call interface
Operating system kernel
Computer hardware

FIGURE 1.1 A layered view of a contemporary computer system.

tools, picture- and video-processing tools, and Internet-related tools such as Web browsers. These applications, which reside at the topmost layer in the diagram, form the application user interface (AUI). The AUI is glued to the operating system kernel via the language libraries and the system call interface. The system call interface comprises a set of functions that can be used by the applications and library routines to execute the kernel code for a particular service, such as reading a file. The language libraries and the system call interface comprise what is commonly known as the *application programmer interface* (API). The kernel is the core of an operating system, where issues like CPU scheduling, memory management, disk scheduling, and interprocess communication are handled. The layers in the diagram are shown in an expanded form for the UNIX operating system in Figure 1.2, where we also describe them briefly.

There are two ways to view an operating system: top down and bottom up. In the bottom-up view, an operating system can be viewed as a software that allocates and deallocates system resources (hardware and software) in an efficient, fair, orderly, and secure manner. For example, the operating system decides how much RAM space is to be allocated to a program before it is loaded and executed. The operating system ensures that only one file is printed on a particular printer at a time, prevents an existing file on the disk

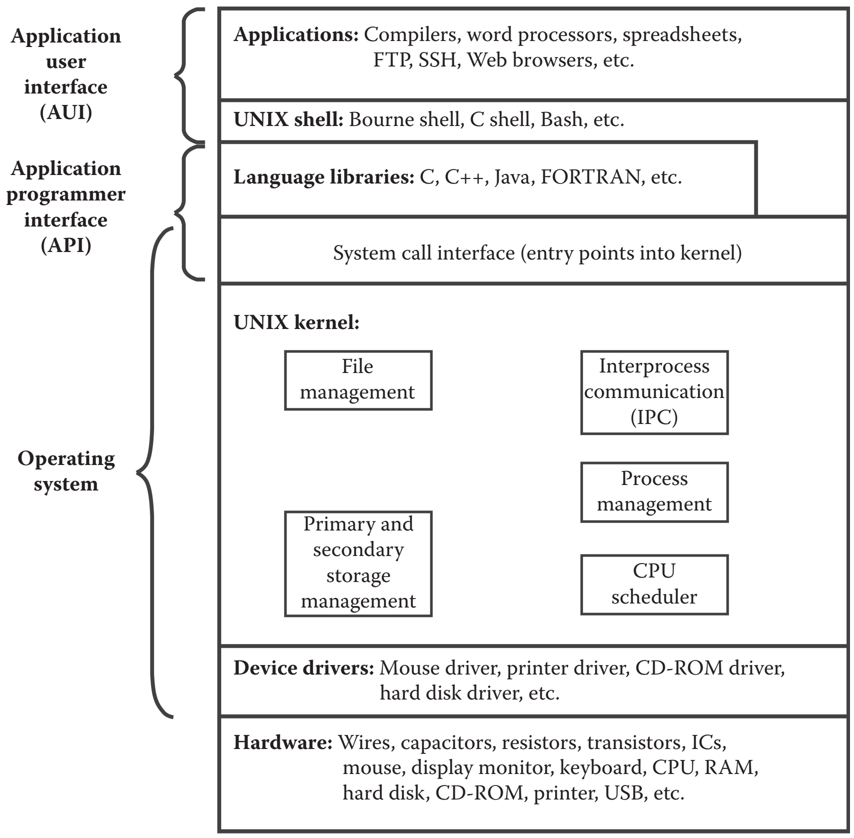

Application user interface (AUI)
Application programmer interface (API)
Operating system
Applications: Compilers, word processors, spreadsheets, FTP, SSH, Web browsers, etc.
UNIX shell: Bourne shell, C shell, Bash, etc.
Language libraries: C, C++, Java, FORTRAN, etc.
System call interface (entry points into kernel)
UNIX kernel:
File management
Interprocess communication (IPC)
Process management
Primary and secondary storage management
CPU scheduler
Device drivers: Mouse driver, printer driver, CD-ROM driver, hard disk driver, etc.
Hardware: Wires, capacitors, resistors, transistors, ICs, mouse, display monitor, keyboard, CPU, RAM, hard disk, CD-ROM, printer, USB, etc.

FIGURE 1.2 Software architecture of the UNIX operating system.

from being accidentally overwritten by another file, and further guarantees that, when the execution of a program given to the CPU for processing has been completed, the program relinquishes the CPU so that other programs can be executed. Thus the operating system can be viewed as a resource manager.

In the top-down view, which we espouse in this textbook, an operating system can be viewed as a piece of software that isolates you from the complications of hardware resources. You therefore do not have to deal with the extremely difficult (and sometimes impossible for most users) task of interacting with these resources. For example, as a user of a computer system, you don't have to write the code that allows you to save your work as a file on a hard disk, use a mouse as a point-and-click device, use a touch screen or touch pad, or print on a particular printer. Also, you do not have to write new software for a new device (e.g., mouse, disk drive, or DVD) that you buy and install in your system. The operating system performs the task of dealing with complicated hardware resources and gives you a comprehensive machine with a simple, ready-to-use interface. This machine allows you to use simple commands to retrieve and save files on a disk, print files on a printer, and play movies from a DVD. In a sense, the operating system provides a virtual machine that is much easier to deal with than the physical machine. You can, for example, use a command such as `cp memo letter` to copy the memo file to the letter file on the hard disk in your computer without having to worry about the location of the memo and letter files on the disk, the structure and size of the disk, the brand of the disk drive, and the number or name of the various drives (floppy, CD-ROM, and one or more hard drives) on your system.

1.3 OPERATING SYSTEM SERVICES

An operating system provides many ready-made services for users. Most of these services are designed to allow you to execute your software, both application programs and program development tools, efficiently and securely. Some services are designed for housekeeping tasks, such as keeping track of the amount of time that you have used the system. The major operating system services therefore provide mechanisms for following secure and efficient operations and processes:

- Execution of a program
- Input and output operations performed by programs
- Communication between processes
- Error detection and reporting
- Manipulation of all types of files
- Management of users and security

A detailed discussion of these services is outside the scope of this textbook, but we discuss them briefly when they are relevant to the topic being presented.

1.4 CHARACTER (COMMAND LINE) VERSUS GRAPHICAL USER INTERFACES

In order to use a computer system, you have to give commands to its operating system. An input device, such as a keyboard, is used to issue a command. If you use the keyboard to issue commands to the operating system, the operating system has a character user interface (CUI), commonly known as the *command line interface*. If the primary input device for issuing commands to the operating system is a point-and-click device, such as a mouse, a touch screen, or a touch pad, the operating system has a graphical user interface (GUI). Most, if not all, operating systems have both character and graphical user interfaces, and you can use either. Some have a command line as their primary interface but allow you to run software that provides a GUI. Operating systems such as DOS and UNIX have CUIs, whereas Mac OS, OS/2, and Microsoft Windows primarily offer GUIs but have the capability to allow a user to enter a DOS- or UNIX-like terminal screen. Although UNIX comes with a CUI as its basic interface, it can run software based on the X Window System (Project Athena, MIT) that provides a GUI interface. Moreover, most UNIX systems now have a state-of-the-art X-based GUI. Mac OS X (Darwin), running on Apple products, is the most well-known GUI-based UNIX system. We discuss the UNIX GUI in Chapter 23.

Although a GUI makes a computer easier to use, it gives you an automated setup with reduced flexibility. A GUI also presents an extra layer of software between you and the task that you want to perform on the computer, thereby making the task slower. In contrast, a CUI gives you ultimate control of your computer system and allows you to run application programs any way you want. A CUI is also more efficient because a minimal layer of software is needed between you and your task on the computer, thereby enabling you to complete the task faster. It is also malleable and gives the user more control. Because many people are accustomed to the graphical interfaces of popular gizmos and applications such as Nintendo and Web browsers, the character interface presents an unfamiliar and sometimes difficult style of communicating commands to the computer system. However, computer science students are usually able to meet this challenge after a few hands-on sessions.

1.5 TYPES OF OPERATING SYSTEMS

Operating systems can be categorized according to the number of users who can use the system at the same time and the number of processes (executing programs) that the system can run simultaneously. These criteria lead to three types of operating systems:

- *Single-user, single-process system*: These operating systems allow only one user at a time to use the computer system, and the user can run only one process at a time. Such operating systems are commonly used for PCs. Examples of these operating systems are earlier versions of Mac OS, DOS, and many of Microsoft's Windows operating systems.

- *Single-user, multiprocess system*: As the name indicates, these operating systems allow only a single user to use the computer system, but the user can run multiple processes simultaneously. These operating systems are also used on PCs. Examples of such operating systems are OS/2, Windows XP Workstation, and batch operating systems. Batch processing is still commonly used in mainframe computers, and most modern operating systems including UNIX, Microsoft Windows, Linux, and Mac OS perform some tasks in batch mode. Even smartphone operating systems including Android and iOS perform tasks in batch mode.

- *Multiuser, multiprocess system*: These operating systems allow multiple users to use a computer system simultaneously, and every user can run multiple processes at the same time. These operating systems are commonly used on computers that support multiple users in organizations such as universities and large businesses. Examples of these operating systems are UNIX, Linux, Windows NT Server, MVS, and VM/CMS.

Multiuser, multiprocess systems are used to increase resource utilization in the computer system by multiplexing expensive resources such as the CPU. This capability leads to increased system throughput (the number of processes finished in unit time). Resource utilization increases because, in a system with several processes, when one process is performing input or output (e.g., reading input from the keyboard, capturing a mouse click, or writing to file on the hard disk), the CPU can be taken away from this process and given to another process—effectively running both processes simultaneously by allowing them both to make progress (one is performing input/output [I/O] and the other is using the CPU). The mechanism of assigning the CPU to another process when the current process is performing I/O is known as *multiprogramming*. Multiprogramming is the key to all contemporary multiuser, multiprocess operating systems. In a single-process system, when the process using the CPU performs I/O, the CPU sits idle because there is no other process that can use the CPU at the same time.

Operating systems that allow users to interact with their executing programs are known as *interactive operating systems*, and the ones that do not are called batch operating systems. Batch systems are useful when programs are run without the need for human intervention, such as systems that run payroll programs. The VMS operating system has both interactive and batch interfaces. Almost all well-known contemporary operating systems (UNIX, Linux, DOS, Windows, etc.) are interactive. UNIX and Linux also allow programs to be executed in batch mode, with programs running in the background (see Chapter 10 for details of "background process execution" in UNIX). Multiuser, multiprocess, and interactive operating systems are known as *time-sharing systems*. In time-sharing systems, the CPU is switched from one process to another in quick succession. This method of operation allows all the processes in the system to make progress, giving each user the impression of sole use of the system. Examples of time-sharing operating systems are UNIX, Linux, and Windows NT Server.

1.6 THE UNIX FAMILY

Years ago the name UNIX referred to a single operating system, but it is now used to refer to a family of operating systems that are offshoots of the original in terms of their user interfaces. Éric Lévénez (www.levenez.com/unix) lists names of over 270 UNIX flavors at the time of writing this book. Some of the members of this family are AIX, BSD, DYNIX, FreeBSD, HP-UX, Linux, MINIX, NetBSD, SCO, Solaris, OpenSolaris, SunOS, System V, XENIX, PC-BSD, OpenBSD, Mac OS X (Darwin), and XINU. In Section 1.8, we give a brief history of some of the most popular and developmentally influential UNIX systems.

1.7 UNIX SOFTWARE ARCHITECTURE

Figure 1.2 shows a layered diagram for a UNIX-based computer system, identifying the system's software components and their logical proximity to the user and hardware. We briefly describe each software layer from the bottom up.

1.7.1 Device Driver Layer

The purpose of the device driver layer is to interact with various hardware devices. It contains a separate program for interacting with each device, including the hard disk driver, floppy disk driver, CD-ROM driver, keyboard driver, mouse driver, touch pad driver, and display driver. These programs execute on behalf of the UNIX kernel when a user command or application needs to perform a hardware-related operation such as a file read that translates to one or more disk reads. The user doesn't have direct access to these programs and therefore can't execute them as commands.

1.7.2 UNIX Kernel

The UNIX kernel layer contains the actual operating system. Some of the main functions of the UNIX kernel, listed in Figure 1.2, are described in this section. In addition, the kernel performs several other tasks for fair, orderly, and safe use of the computer system. These tasks include managing the CPU, printers, and other I/O devices. The kernel ensures that no user process takes over the CPU forever, that multiple files are not printed on a printer simultaneously, and that a user cannot terminate another user's process.

1.7.2.1 Process Management

This part of the kernel manages processes in terms of creating, suspending, resuming, and terminating them, and maintaining their states. It also provides various mechanisms for processes to communicate with each other and schedules the CPU to execute multiple processes simultaneously in a time-sharing system. *Interprocess communication* (IPC) is the key to the client–server-based software that is the foundation for Internet applications, including Web browsing (HTTP), file transfer (FTP), and remote login (SSH). The UNIX system provides three primary IPC mechanisms/channels:

- *Pipe*: Two or more related processes running on the same computer can use a pipe as an IPC channel. Typically, these processes have a parent–child or sibling relationship. A pipe is a temporary channel that resides in the main memory and is created by the kernel, usually on behalf of the parent process.
- *Named pipe*: A named pipe, also known as a FIFO, is a permanent communication channel that resides on the disk and can be used for IPC by two or more related or unrelated processes running on the same computer.

- *BSD socket*: A BSD socket is also a temporary channel that allows two or more processes in a network (or on the Internet) to communicate, although processes on the same computer can also use them. Sockets were originally a part of the BSD UNIX only, but they are now available on almost every UNIX system. Internet software such as Web browsers, File Transfer Protocol (FTP), Secure Shell (SSH), and electronic mailers are implemented by using sockets. AT&T UNIX has a similar mechanism called the Transport Layer Interface (TLI).

We discuss these mechanisms of IPC in detail under "UNIX System Programming" in Chapters 18 through 21.

1.7.2.2 File Management

This part of the kernel manages files and directories, also known as folders. It performs all file-related tasks, including file creation and removal, directory creation and removal, setting access privileges on files and directories, and maintaining their attributes, such as file size. A file operation usually requires manipulation of a disk. In a multiuser system, a user must never be allowed to manipulate a disk directly because it contains files belonging to other users, and user access to a disk poses a security threat. Only the kernel must perform all file-related operations, such as file removal. Also, only the kernel must decide where and how much space to allocate to a file.

1.7.2.3 Main Memory Management

This part of the kernel allocates and deallocates RAM in an orderly manner so that each process has enough space to execute properly. It also ensures that part or all of the space allocated to a process does not belong to some other process. The space allocated to a process in the memory for its execution is known as its *address space*. The kernel ensures that no process accesses an area of memory that does not belong to its address space. The kernel maintains areas in the main memory that are free to be allocated to processes. The kernel code that performs this task is called the *free space manager*. When a program is to be loaded in the main memory, the free space manager allocates adequate space for it and the *loader* loads the program into this space. The kernel also records where all the processes reside in the memory so that, when a process tries to access main memory space that does not belong to it, the kernel can terminate the process and give a meaningful message to the user. When a process terminates, the kernel deallocates the space allocated to the process and puts it back in the free space pool so that it can be reused.

1.7.2.4 Disk Management

The kernel is also responsible for maintaining free and used disk space and for the orderly and fair allocation and deallocation of disk space. It decides where and how much space to allocate to a newly created file. The kernel code that performs this task is known as the *disk storage manager*. Also, the kernel performs *disk scheduling*, deciding which request to serve next when multiple requests for file read, write, and so on, arrive for the same disk.

1.7.3 System Call Interface

The system call interface layer contains entry points into the kernel code. Because the kernel manages all system resources, any user or application request that involves access to any system resource must be handled by the kernel code. But user processes must not be given open access to the kernel code for security reasons. So that user processes can invoke the execution of kernel code, UNIX provides several openings, or function calls, into the kernel, known as *system calls*. There are numerous system calls that deal with the manipulation of processes, files, and other system resources. These calls are well tested, and most of them have been used for several years, so their use poses much less of a security risk than if any user code were allowed to perform the task.

1.7.4 Language Libraries

A *language library* is a set of prewritten and pretested functions in a programming language available to programmers for use with the software that they develop. The availability and use of libraries saves time because programmers do not have to write these functions from scratch. This layer contains libraries for several languages, including C, C++, C#, Java, Perl, and Python. For the C language, for example, there are several libraries, including a string library (which contains functions for processing strings, such as a function for comparing two strings) and a math library (which contains functions for mathematical operations, such as finding the cosine of an angle).

As we stated earlier in this chapter, the libraries and system call interface form what is commonly known as the API. In other words, programmers who write software in a language such as C can use in their code the prewritten functions available in the various C libraries and system calls.

1.7.5 UNIX Shell

The *UNIX shell* is a program that starts running when you log on and interprets the commands that you enter. The most popular shells are the Bourne shell (`sh`), Bourne Again shell (`bash`), C shell (`csh`), TC shell (`tcsh`), and Korn shell (`ksh`). We show the usage of shell commands and shell scripts (see Chapters 12 through 15) in Bourne and C shells.

1.7.6 Applications

The applications layer contains all the applications (tools, commands, and utilities) that are available for your use. A typical UNIX system contains hundreds of applications; we discuss the most useful and commonly used applications throughout this textbook. When an application that you're using needs to manipulate a system resource (e.g., reading a file), it needs to invoke some kernel code that performs the task. An application can find the appropriate kernel code to execute in one of two ways: (1) by using a proper library function and (2) by using a system call. Library calls constitute a higher-level interface to the kernel than system calls, which makes library calls a bit easier to use. However, all library calls eventually use system calls to begin execution of the appropriate kernel code. Therefore, the use of library calls in software results in slightly slower execution. A detailed discussion of language libraries and system calls is, generally, beyond the scope of this textbook. However, we discuss and show the use of several library calls and system calls in Chapter 16 on Python and Chapters 18 through 21 on UNIX system programming.

The user can use any command or application that is available on the system. As we mentioned earlier in this chapter, this layer is commonly known as the AUI.

1.8 DEVELOPMENT OF THE UNIX OPERATING SYSTEM

How has UNIX achieved the status and position in the marketplace it has now? Because it is a multiuser, multiprocess operating system that can run on the X86 architectures of very small- to very large-scale hardware used by most computer users in the world. That means it is used by casual, individual users on their home computers, all the way up to 60 processor servers in a cloud configuration at a commercial facility. It can link your home computer to the Internet with a standard browser like Opera or Firefox. Also, the vast majority of Internet servers run on UNIX or Linux machines.

What really are the differences between the three major families of UNIX? Basically, the kernels and their APIs, the file systems, the device driver bases, and to some extent the desktop management systems they use. And accordingly, in reality there are only two true UNIX operating system families now: the BSD family, exemplified in our book by the FreeBSD offshoot named PC-BSD, and the Solaris family. Both families have the distinguishing feature of booting from and including the Z File system (ZFS) in their kernel. At the time of writing this book, none of the other NIX-like systems (Linux, OS X) have this feature.

The people who use the UNIX operating system are application developers, systems analysts, web programmers, system administrators, business managers, academicians, and people who just want to read their e-mail. From its earliest inception in 1969 as a laboratory research tool, it was further developed in the academic community and then endorsed for commercial uses. Today's UNIX has an underlying functionality that is complex but easy to learn because of its GUI, and extensible yet easily customized to suit a user's style of computing. The GUI is comparable to those on Windows or OS X machines. One of the primary keys to understanding its longevity, and its heterogeneous appeal, is to study the history of its evolution, which follows.

1.8.1 Beginnings

Before we describe the evolution of UNIX, first we have to ask, why is this operating system so "friendly" and accommodating? Part of the answer is that this ever-evolving operating system, which is accepted and used throughout the world, was developed in response to the needs and activities of a very heterogeneous community of computer users. It grew, changed, and improved because of the work and cooperation of many diverse, and sometimes opposing, individuals and groups.

UNIX continuously grew, changed, and improved alongside the development of computer hardware, software applications, networking, and other components of the "computer revolution." The UNIX project started as a personal and subjective endeavor but exploded into a universal and generic technical tool. Thus its various audiences must have found some basic advantages in this tool—particularly the largest audience, common users. Separating the influences of these various user groups in the development of UNIX is difficult. Moreover, because the system is fundamentally an open software system—that is, the source code is freely distributed among the community of users—its evolution has been shaped to some extent by a populist mindset. For example, development resource and source code repositories such as GitHub expedite this development model in the twenty-first century. It will continue to be shaped as such in the future, thanks to organizations like the Open Software Foundation, and because of the pervasive use of the Internet in social life, academic settings, and in business and professional settings.

It is the underlying core functionality of UNIX that brings together its diverse audiences into a community—not so much in the sociological sense, but more in an independent, DIY, intellectual sense. As you delve into the subject matter of this textbook, you might wonder where you fit into the UNIX community and how its functionality might be adapted for your uses. Essentially, it is the style of your interaction with the computer that will be the most important, invigorating, and critical aspect of your work with the UNIX operating system.

The development of other contemporary operating systems is motivated and informed by completely different forces and bases (primarily commercialization) than those that motivated the inception and development of UNIX (primarily a user-friendly, text-based operating system). The history of UNIX is a record of how a system should be developed, regardless of how you believe that system should be structured, how you think it should function (whatever your user perspective), and who you believe should control that development.

Figure 1.3 describes the three main branches of UNIX systems as they were developed from 1969 to the present. The approximate dates of the development of milestone versions in each of the branches are shown on the left.

The UNIX Support Group (USG), UNIX System Development Laboratory (USDL), and UNIX System Laboratories (USL) were commercial spin-offs of AT&T. The UNIX Programmer's Work Bench (PWB) was distributed initially through the USG.

In the mid-1960s, Bell Laboratories began a collaborative effort to develop a multiuser operating system known as MULTICS. One of the biggest drawbacks inherent in the functionality of this new operating system was the complexity of the software and hardware required to accomplish simple tasks for multiple users. Following the failure of the MULTICS project, Ken Thompson, Dennis Ritchie, and others at Bell Laboratories developed a multiuser operating system called UNIX, which first ran on a DEC PDP-7 computer and later was ported to a PDP-11 computer. One of the features of UNIX that distinguished it from MULTICS was that it allowed processes to be created easily by a single user.

The most important historical development in the early 1970s was the recoding of most of the operating system in the high-level programming language, C. At that time, most operating system programs were written in a low-level programming language, known as an *assembly language*, which was specifically tailored to the architecture of the processor that a particular make of computer used. Thus, an operating system written in a low-level language was not portable between computers with different processors made by different manufacturers. Written in C, UNIX was very portable. Also, C, as well as other high-level languages, is much easier to program with than assembly language, which is characteristically difficult.

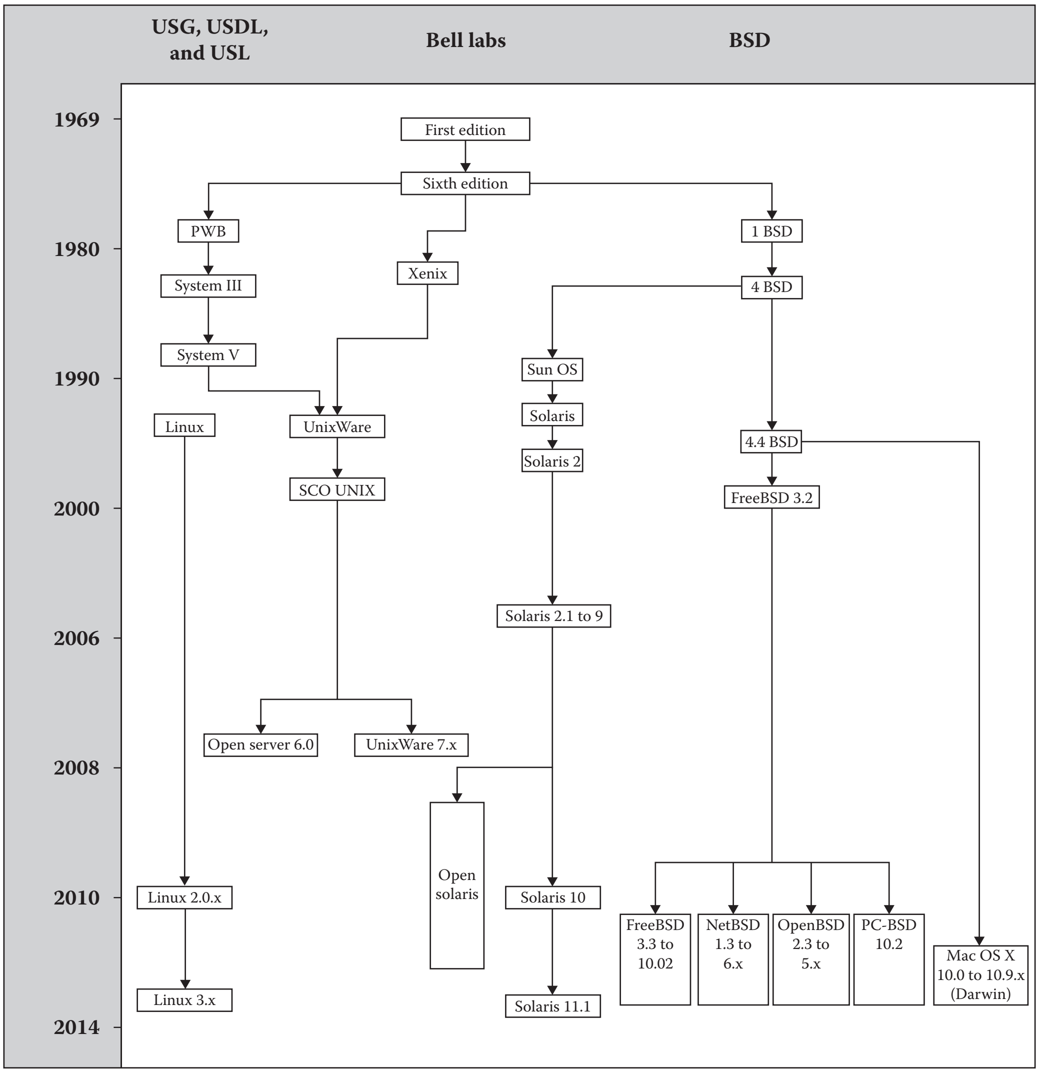


FIGURE 1.3 Schematic UNIX time line.

1.8.2 Research Operating System

Bell Laboratories controlled the research systems versions of UNIX, known as versions 1 through 6. These versions had three important characteristics:

1. The UNIX system was continually developed and written in C, with only a small subset of the code tailored to a target processor.
2. Releases were distributed as C source code, which could be easily modified and improved upon to add functionality by those who obtained any of the research versions of the system.

3. The design of the system allowed users to run multiple processes concurrently and to connect these processes with IPC channels, including pipes, FIFOs, and sockets, as discussed in Section 1.7.2. We present the implementation of this design aspect in Chapter 9.

1.8.3 AT&T System V

In response to the changing business environment in the early 1980s, Bell Laboratories/AT&T licensed further releases of UNIX as System III and finally as System V, starting in 1983. This main branch of UNIX continued to be developed, as shown in Figure 1.3, through System V, Release 4 (SVR4), when it again diverged and evolved to survive as SCO UNIX in the mid- to late 1990s. Currently, there are four major System V–based UNIX systems: AIX 7.x, OpenServer 6.x, UnixWare 7.x, Solaris 11.x, OpenSolaris and its variants, and HP-UX 11i v3.

1.8.4 Berkeley Software Distributions

The University of California at Berkeley initiated and maintained the development of UNIX along its second main branch throughout the 1980s and into the 1990s. Contractual agreements made the operating system freely available to universities, so these releases contributed in large part to the popularization of UNIX. These versions were released as Berkeley Software Distributions (BSD), 3BSD, and 4BSD–4.4BSD. Most recently, BSD UNIX survives as FreeBSD, and its offshoot PC-BSD, OpenBSD, and NetBSD. Today, the most popular variants of BSD UNIX are FreeBSD 10.x, OpenBSD 5.x, NetBSD 6.x, and Mac OS X 10.9.x (Darwin).

In this book, we show example commands primarily under PC-BSD. Where appropriate, we also discuss features of Solaris.

1.8.5 History of Shells

The development of the shell as a UNIX utility parallels the development of the system itself. Steven R. Bourne wrote the first commercially available shell, the Bourne shell. Available in the seventh edition in 1979, it is the default shell on many System V versions. The C shell, written in the late 1970s primarily by Bill Joy, was made available soon after in 2BSD. When introduced, it provided a C program–like programming interface for writing shell scripts. Following the development of the C shell, the Korn shell was introduced officially in SVR4 in 1986. Written by David Korn of Bell Laboratories, it included a superset of Bourne shell commands but had more functionality. It also included some useful features of the C shell. We discuss the development history of the UNIX shell in a bit more detail in Chapter 2.

The three major shells have slightly different features and command sets. In this textbook, we discuss common features and command sets for all UNIX shells and versions. Whenever we discuss a feature or command that is particular to a shell or version, we state that specifically.

1.8.6 Current and Future Developments

Probably the most exciting and challenging current UNIX development for all other flavors of UNIX (besides Solaris and FreeBSD) focuses on the incorporation of the ZFS into the kernel, and having the boot disk use ZFS. ZFS was developed by Sun Microsystems in the years prior to its incorporation into the Solaris family in 2006. It is now the standard file system in only Solaris (and its noncommercial equivalent, OpenIndiana) and FreeBSD (and its offshoot PC-BSD). Since the purchase of Sun by Oracle, the source development of ZFS has proceeded basically along two branches: the Oracle Solaris branch and the open branch.

Another major challenge on the horizon for UNIX systems is the incorporation of systemd into the kernel. Currently, systemd is a suite of system management daemons, libraries, and utilities designed as a central management and configuration platform for Linux. systemd is used on a majority of the current implementations and official releases of the Linux kernel. It is a Linux init system (the process called on by the Linux kernel to initialize the user space during the Linux startup process and manage all processes afterwards), thus replacing the UNIX System V and BSD-style `init` daemon. The name systemd adheres to the convention of making daemons easier to distinguish by having the letter d as the last letter of the filename. Whether or not systemd will be incorporated into the UNIX kernel remains to be seen.

Finally, the replacement of the X Window System protocol by various other software systems promises to yield a smaller, more effective GUI system. Wayland is a protocol that specifies the communication between a display server (called a Wayland compositor) and its clients, as well as a reference implementation of the protocol in C.

1.9 VARIATIONS IN UNIX SYSTEMS

As shown in Figure 1.3, the development of the UNIX systems proceeded along three main branches from a single core. Many of the branches' divergences and similarities were caused by the Bell Laboratories and AT&T legal licensing arrangements during the 1970s and 1980s. The primary advantage of the divergences was a command- and function-rich operating system in each of the branches. The early Bell Labs releases were copied and distributed freely as source code, which academic and commercial users could easily modify to suit their hardware and software. Such adaptations led to a proliferation of ways in which various aspects of the operating system evolved. Even the later releases of System V and BSD could be modified easily via accommodations provided by the vendor of the operating system version, even if the source code was not available. Many of the later releases were compatibility releases meant to provide uniformity between any particular implementation and its perceived competitors. The important contribution of these compatibility releases and their offshoots is a helpful amount of homogeneity, regardless of whether you use a modern derivative of SVR4, Solaris, 4.4BSD, Linux, or Apple OS X.

Divergence has the drawback that programs and even commands that work on one version fail to work on another, thus defeating the inherent strength of user-friendliness of the system itself. Attempts have been made to standardize UNIX—for example, via the IEEE Portable Operating System Interface (POSIX). This software standard not only covers UNIX, but also in particular specifies program operation and user interfaces, leaving their implementations to the developer. Several standards have been adopted, and more have been proposed. For example, adopted POSIX standards specify the shell and utility standardization.

By far the most inclusive and wide-ranging standardization mechanism is the Single UNIX Specification (SUS). This is a family of standards for computer operating systems, compliance with which is required to qualify for the name UNIX. The core specifications of the SUS are developed and maintained by the Austin Group, which is a joint working group of IEEE, ISO/IEC JTC 1/SC 22, and the Open Group.

SUMMARY

An operating system is software that runs on the hardware of a computer system to manage its hardware and software resources. It also gives the user of the computer system a simple, virtual machine that is easy to use. The basic services provided by an operating system offer efficient and secure program execution, I/O operations, communication between processes, error detection and reporting, and file manipulation.

Operating systems are categorized by the number of users that can use a system at the same time and the number of processes that can execute on a system simultaneously: single-user single-process, single-user multiprocess, and multiuser multiprocess operating systems. Furthermore, operating systems that allow users to interact with their executing programs (processes) are known as *interactive systems*, and those that do not are called batch systems. Multiuser, multiprocess interactive systems are known as *time-sharing systems*, of which UNIX is a prime example. The purpose of multiuser, multiprocess systems is to increase the utilization of system resources by switching them among concurrently executing processes. This capability leads to higher system throughput, or the number of processes finishing in unit time.

In order to use a computer system, the user issues commands to the operating system. If an operating system accepts commands via the keyboard, it has a CUI. If an operating system allows users to issue commands via a point-and-click device such as a mouse, it has a GUI. Although UNIX comes with a CUI as its basic interface, it can run software based on the X Window System (Project Athena, MIT) that provides a GUI. Most UNIX systems now have both interfaces. Mac OS X (Darwin), running on Apple products, is the most well-known GUI-based UNIX system.

A computer system consists of several hardware and software components. The software components of a typical UNIX system consist of several layers: applications, shell, language libraries, system call interface, UNIX kernel, and device drivers. The kernel is the main part of the UNIX operating system and performs all the tasks that deal with allocation and deallocation of system resources. The shell and applications layers contain what is commonly known as the AUI. The language libraries and the system call interface contain the API.

The historical development of UNIX is characterized by an open systems approach, whereby the source code was freely distributed among users. Development of many versions of UNIX progressed along three main branches. Two of these branches, Oracle Solaris and FreeBSD, can best be characterized as commercial and academic. Compatibility releases of various versions have been aimed at standardizing the system. The POSIX and the SUS are related standardization efforts. Two exciting and challenging new developments in the future of true UNIX systems are incorporation of ZFS into the kernel, and the replacement of the X Window System protocol with systems such as Wayland.

QUESTIONS AND PROBLEMS

1. What is an operating system?
2. What are the three types of operating systems? How do they differ from each other?
3. What is a time-sharing system? Be precise.
4. What are the main services provided by a typical contemporary operating system? What is the basic purpose of these services?
5. List one advantage and one disadvantage each for the CUI and the GUI.
6. What is the difference between a CUIs and GUIs? What is the most popular GUI for UNIX systems? Where was it developed?
7. What comprises the API and the AUI?
8. What is an operating system kernel? What are the primary tasks performed by the UNIX kernel?
9. What is a system call? What is the purpose of the system call interface?
10. If you access a UNIX system with the `ssh` command, write down the exact step-by-step procedure you go through to log on and log off. Include as many descriptive details as possible in this procedure so that if you forget how to log on, you can always refer back to this written procedure.
11. What is a shell? Name the most popular UNIX shells. Log on to your UNIX computer system and note the shell prompt being used.
12. How can you tell which variant from the main branches of UNIX (see Figure 1.3) is being used on the computer system that you log on to?
13. If you were designing a POSIX standard, what would you include in it? You might want to research the already adopted and proposed standards before answering this question.
14. If you were designing an SUS standard, what would you include in it? You might want to research the already adopted SUS standards, presented briefly in the chapter and online, before answering this question.
15. What system was the immediate predecessor of UNIX? Where was this predecessor and UNIX itself initially developed, and by whom?
16. Name the major versions and the three main branches of UNIX development. Which was the commercial branch? Which was the academic branch?

17. What three important characteristics of UNIX during its early development helped popularize it? Explain how these characteristics apply to you as a UNIX user, whatever your perspective.
18. Name the two most popular UNIX systems that are the basis of most UNIX systems. Where were they developed?
19. Trace the history of UNIX by browsing the Web. How many UNIX systems have been developed so far? How many non-UNIX systems have been developed? What is the most popular UNIX system for PCs? Why do you think it is so popular?
20. Name five popular members of the UNIX family. What is the name of your UNIX system?
21. In the late 1960s and early 1970s, the Digital Equipment Corporation (DEC) was a key player in the development of time-sharing systems. Browse the Web and find an article on RSTS, an operating system developed at DEC. What was its full name? What machines did it run on? What were its key features?

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

CHAPTER 2

A "Quick Start" into the UNIX Operating System

Objectives

- To introduce the UNIX character user interface (CUI) and show the generic structure of UNIX commands
- To describe how to connect and log on to a computer running the UNIX operating system, particularly PC-BSD and Solaris
- To explain how to manage and maintain files and directories
- To show where to get online help for UNIX commands
- To demonstrate the use of a beginner's set of utility commands
- To describe what a UNIX shell is
- To describe briefly some commonly used shells
- To cover the basic commands and operators

  `alias, biff, cal, cat, cd, chsh, cp, csh, echo, exit, hostname, login, logout, lp, lpr, ls, man, mesg, mkdir, more, mv, passwd, PATH, pg, pwd, rm, rmdir, set, ssh, su, sudo, talk, telnet, unalias, uname, whatis, whereis, who, whoami, write`

2.1 INTRODUCTION

To start working productively in UNIX, the beginner needs to know eight sequential topics, in the order presented as follows:

1. How to type a syntactically correct command on the UNIX command line. One of the most useful modes of interaction with the UNIX system uses text-based, typed commands.

2. How to log in and log out of a computer running UNIX, using one of the standard methods we show. UNIX allows users to enter the operating system autonomously, do a combination of text- and graphics-oriented operations, and exit gracefully.
3. How to maintain and organize files in the file structure. Creating a tree-like structure of folders (also called directories), and storing files in a logical fashion in these folders, is critical to working efficiently in UNIX.
4. How to get help on commands and their usage. In the command-based CUI environment, being able to find out, in a quick and easy way, how to use a command correctly is imperative to working efficiently.
5. How to execute a small set of essential utility commands to set up or customize your working environment. Once a beginner is familiar with the right way to construct file maintenance commands, adding a set of utility commands makes each session more productive.
6. The essential ways to work with UNIX shells, what they are, and how to find out what shell is running when you log in.
7. Ways to change your shell, and what shell environmental variables are.
8. What shell metacharacters are.

To use this chapter successfully as a springboard into the remainder of the book, you should read and follow the instructions, in the order presented. Each chapter builds on the information that precedes it, and will give you the concepts, command tools, and methods to program in the UNIX operating system. In this chapter, the major commands are defined with an abbreviated syntax description, which will clarify general components for the remainder of the textbook as follows:

SYNTAX

The exact syntax of how a command, its options, and its arguments are typed on the command line

Purpose: The specific purpose of the command
Output: A short description of the results of executing the command
Commonly used options/features: A listing of the most popular and useful options and option arguments

2.2 THE STRUCTURE OF A UNIX COMMAND

Because UNIX is reliant on both a graphical and a text-based CUI, correctly typed syntax is critical to ensure subsequent correct execution of commands.

After a user successfully logs on to a UNIX computer, a shell prompt, such as the `$` character, appears on the screen. The shell prompt is simply a message from the computer system to say that it is ready to accept keystrokes on the command line that directly follows the prompt. The general syntax, or structure of a *single* command (as opposed to a command line that may have *multiple* commands typed on the same line, separated with input and output redirection characters) as it is typed on the command line is as follows:

```
$ command [[-]option(s)] [option argument(s)] [command argument(s)]
```

where:

$ is the command line or shell prompt from the computer;
anything enclosed in **[ ]** is not always needed;
command is the name of the valid UNIX command for that shell in lowercase letters;
[-option(s)] is one or more modifiers that change the behavior of command;
[option argument(s)] is one or more modifiers that change the behavior of **[-option(s)]**; and
[command argument(s)] is one or more objects that are affected by **command**.

Note the following seven essentials:

1. A space separates command, options, option arguments, and command arguments, but no space is necessary between multiple option(s) or multiple option arguments.
2. The order of multiple options or option arguments is irrelevant.
3. A space character is optional between the option and the option argument.
4. Always press the `<Enter>` key to submit the command for interpretation and execution.
5. Options may be preceded by a single hyphen `-` or two hyphens, `--`. No space character between hyphen(s) and option(s).
6. A small percentage of commands (like `whoami`) take no options, option arguments, or command arguments.
7. Everything on the command line is case sensitive!

The following are examples of commands typed on the UNIX command line after the $ prompt, and illustrate some of the variations of the correct syntax for a single command that may have options and arguments:

```
$ ls
$ ls -la
$ ls -la m*
$ lpr -Pspr -n 3 proposal.txt
```

The first example contains only the command. The second contains the command `ls` and two options, `l` and `a`. The third contains the command `ls`, two options, `l` and `a`, and a command argument, `m*`. The fourth contains the command `lpr`, two options, `P` and `n`, two option arguments, `spr` and `3`, and a command argument, `proposal.txt`.

You must also use the following rule of thumb: If the command executes properly, then you are returned to the shell prompt; if it does not execute properly, then you get an error message displayed on the command line, and then you are returned to the shell prompt. For example, if you type `xy` on the command line and then press `<Enter>`, usually you will get an error message saying that no such command can be found, and you are returned to the shell prompt so that you can keystroke a valid command.

This rule of thumb does not ensure that what you wanted to achieve by typing the syntactically correct command on the command line will be achieved. That is, you could execute a command and get no error messages. But the command may not have done the things you wanted it to do, simply because you used it with the wrong options or command arguments.

2.3 LOGGING ON AND LOGGING OFF

How can you log on to a UNIX computer and then gracefully leave?

Using one of these general ways, or a hybrid version of them:

- *Stand-alone*: Use a stand-alone system, such as the PC-BSD or Solaris systems we use throughout this book, where UNIX is the only operating system on the hardware.
- *Remote*: Connect to a remote computer running UNIX from a computer running UNIX or another operating system.
- *Virtual*: Start UNIX as a guest operating system in a virtual environment, such as VirtualBox or VMware, while another operating system is the host system.

These general ways are the first step a user takes in a typical UNIX session: gaining access to a UNIX system properly in an autonomous way.

A more detailed description of these ways follows:

1. *Stand-alone*: This way, the most common case and the methodology we deploy in the rest of this book with PC-BSD and Solaris, involves sitting at a computer that can function completely on its own. This does not mean that the stand-alone computer is not hooked up to a local area network (LAN), intranet, or the Internet.

   Rather, the users' connection to UNIX is dedicated to a single user at a time (or possibly many autonomous users that log on to the same system individually at different times) sitting at the computer and logging on to use UNIX on that hardware platform exclusively.

2. *Remote*: There are several variations of using this way. Here are just two possible scenarios:

   a. You sit at a computer that acts like the traditional *terminal* connected to a mainframe computer. This could also be a *thin client* (a minimally configured and capable device) connected to a server. It is connected by a high-speed communications link to another single computer or multiple computers that are all interconnected with a LAN or the Internet. At the terminal, and the console or command window that appears on its screen, your interface with the operating system runs on a single, or even multiple, other computer(s). This is a shared resource method, where several users on many different terminals can share a single UNIX system.

b. You sit at a stand-alone computer, and via software such as PuTTY, Secure Shell (SSH), or SSH X Windows forwarding (a variant of TCP port forwarding), you connect to another system over a high-speed telecommunications link. The PuTTY or SSH software then becomes your *graphical connection*, allowing you to log on and use a remote computer or system that is running UNIX. This is usually a shared resource method, where several users on many different remote computers can share a single UNIX system.

3. *Virtual*: You have a UNIX or NIX-like operating system such as LINUX, OS X, or another operating system installed and running the computer you are sitting in front of, and you have installed a virtual environment such as VirtualBox or VMware on that computer. Then, when you want to use a UNIX system, you simply switch environments so that the UNIX system in the virtual environment is what you are using to interface with the computer hardware.

We do not cover virtual connections in this chapter, but in Chapter 25, "Virtualization Methodologies," we cover virtual environments such as VirtualBox.

In the following subsections, we present three practical, useful, easy, and popular ways of connecting and logging on and off a computer running the UNIX system, as outlined in Section 2.3.

The three ways we show are:

1. Stand-alone login and logout for PC-BSD and Solaris.
2. Remote login via the PuTTY program from a computer running Microsoft Windows to a UNIX computer running PC-BSD.
3. Remote login via an SSH client from a UNIX client computer running PC-BSD UNIX to another remote UNIX host computer.

What is common to all three of these ways is that your first task is to identify yourself correctly as a valid and autonomous user to the UNIX system. Doing so involves typing in a valid username, or login name, consisting of a string of valid characters. You then have to type in a valid password for that username.

Before proceeding with the remainder of this chapter, you should determine which one of the preceding three ways you will use to log in to a UNIX system, and then select from the three following sections that give details on how to use that way correctly. If you cannot determine this on your own, get help from your instructor or the system administrator at your site. Be aware that you may have to use a hybrid way of logging in and out.

2.3.1 Stand-Alone Login Connection to PC-BSD and Solaris

The login and logout procedures shown in this section are standard and vary only slightly between all UNIX and UNIX-like systems. This way assumes that someone has either logged out gracefully or rebooted the computer before you got to it, but has not shutdown the system.

In this section, it is assumed that you are logging on to an already-running computer with PC-BSD UNIX or Solaris as the operating system. As previously stated, when you log in, identifying yourself to the UNIX system is your first task. Doing so involves typing in a valid username, or login name, consisting of a string of valid characters. Then you type in a valid password for that username.

Be aware that when typing on the command line, UNIX is case sensitive!

2.3.1.1 PC-BSD Login and Logout

The login window to PC-BSD is shown in Figure 2.1.

In the login window, you should keystroke your username in the username field, and then by default keep the desktop manager as KDE. If other window or desktop management systems were previously installed on the computer, you can choose one of those in the login window.

In the password field of the login window, type in your password. Finally, click on the right-facing arrow as seen in Figure 2.1, and you will be logged in. On PC-BSD, the KDE desktop management system appears on screen by default.

To gracefully terminate your connection with the computer running PC-BSD, make the Kickoff Application Launcher menu choices **Leave>Log Out**.

2.3.1.2 Solaris Login and Logout

The login windows to Solaris are shown in Figures 2.2 and 2.3.

Two sequential login windows appear, allowing you to type in your username in the first and password in the second. On Solaris, after you log in, the Gnome desktop management interface appears on screen by default.

To log out, make the pull-down **System>Logout** menu choice.

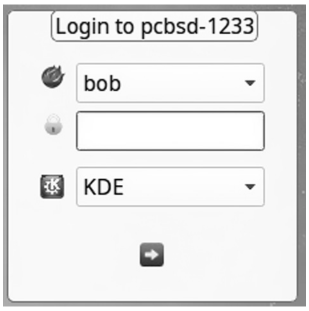

Login to pcbsd-1233
bob
KDE

FIGURE 2.1 Stand-alone login window on PC-BSD.

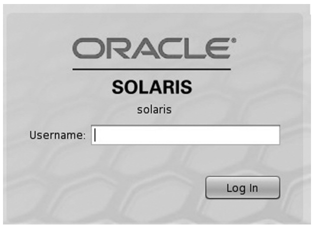


FIGURE 2.2 Stand-alone login window #1 on Solaris.

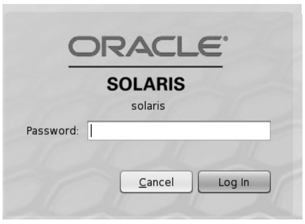


FIGURE 2.3 Stand-alone login window #2 on Solaris.

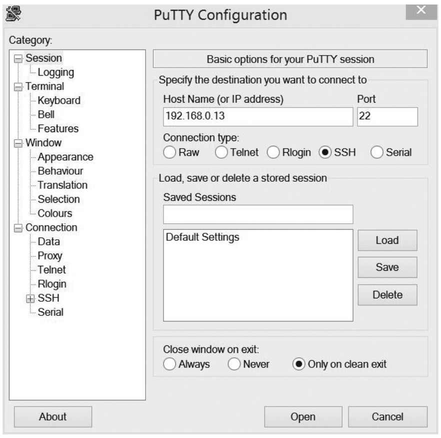
2.3.2 Connecting via PuTTY from a Microsoft Windows Computer

In this section, we make these basic assumptions:

1. That you are sitting at a computer running Microsoft Windows, and trying to connect and log on to a computer running the UNIX operating system.
2. On your Microsoft Windows computer, you are connected to the Internet, or an intranet where you know the Internet Protocol (IP) address of the UNIX computer you want to log on to.
3. You have downloaded and installed the PuTTY program on your Microsoft Windows computer or the system administrator has done so for you. The details of downloading this software and installing it are not given here. At the time of writing, the most current download site for the PuTTY program was: `http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html.`
4. You are using PuTTY to make an SSH connection to a UNIX computer.
5. You know a valid username and password pair that will allow you to log in to the UNIX computer.

Once you execute the PuTTY program, you use the valid username/password pair, and then you can type commands into a console window or terminal screen. What you type in is shown as follows in **bold** text and is always followed by pressing the `<Enter>` key on the keyboard.

To begin, on the Microsoft Windows computer, double click on the PuTTY program icon, or from the **Start Menu>Programs** submenu, and choose PuTTY. When the PuTTY program first launches, the PuTTY configuration dialog window opens on screen, similar to Figure 2.4.

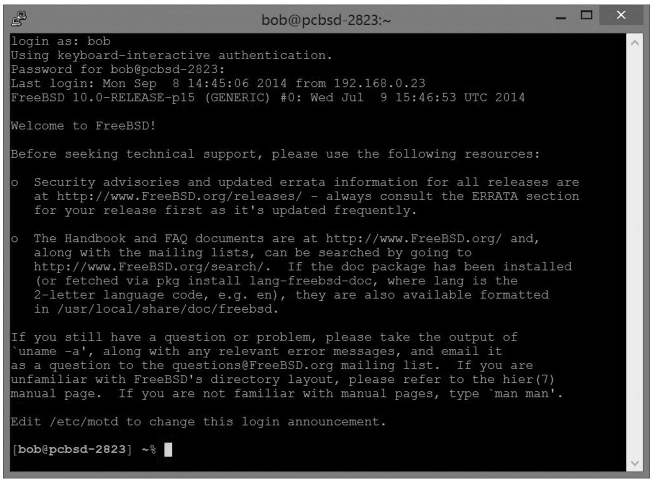


Specify the destination you want to connect to
Host Name (or IP address): 192.168.0.13
Port: 22
Connection type: Raw Telnet Rlogin [✓ SSH] Serial
Load, save or delete a stored session
Saved Sessions:
Default Settings
Close window on exit: Always Never [✓ Only on clean exit]

FIGURE 2.4 PuTTY configuration dialog window.

From the PuTTY configuration window, you can modify several of the parameters that control your interactive session with a UNIX system. Almost all of these parameters can be left at their defaults. The only two things that most users will need to do in this configuration window is type the host name (or IP address) of the UNIX computer they are trying to connect and log in to, and click the protocol button for SSH, as seen in Figure 2.4. The port number is automatically set at 22 if you click on the `SSH` button for "Connection type." You need to know what the host name or IP address of the UNIX computer you want to log on to is. Then click on the `Open` button and a console window will open on screen, as seen in Figure 2.5, thus allowing you to log in to the UNIX computer.

As previously stated, in the process of logging in, identifying yourself to the UNIX system is your first task. Doing so involves typing in a valid *username*, or *login name*, consisting of a string of valid characters. You then type a valid *password* for that username. There are both valid and invalid characters that you can use in both your username and password.



```
login as: bob
Using keyboard-interactive authentication.
Password for bob@pcbsd-2823:
Last login: Mon Sep  8 14:45:06 2014 from 192.168.0.23
FreeBSD 10.0-RELEASE-p15 (GENERIC) #0: Wed Jul  9 15:46:53 UTC 2014

Welcome to FreeBSD!

Before seeking technical support, please use the following resources:

o   Security advisories and updated errata information for all releases are
    at http://www.FreeBSD.org/releases/ - always consult the ERRATA section
    for your release first as it's updated frequently.

o   The Handbook and FAQ documents are at http://www.FreeBSD.org/ and,
    along with the mailing lists, can be searched by going to
    http://www.FreeBSD.org/search/.  If the doc package has been installed
    (or fetched via pkg install lang-freebsd-doc, where lang is the
    2-letter language code, e.g. en), they are also available formatted
    in /usr/local/share/doc/freebsd.

If you still have a question or problem, please take the output of
`uname -a', along with any relevant error messages, and email it
as a question to the questions@FreeBSD.org mailing list.  If you are
unfamiliar with FreeBSD's directory layout, please refer to the hier(7)
manual page.  If you are not familiar with manual pages, type `man man'.

Edit /etc/motd to change this login announcement.

[bob@pcbsd-2823] ~%
```

FIGURE 2.5 PuTTY login window.

See your system administrator or instructor to find out what these characters are on the UNIX system you want to log in to, if they have not already told you what they are.

As shown in Figure 2.5, in response to the `login:` prompt, you type in your username on the UNIX system, and then press `<Enter>` on the keyboard. In our case the username is **bob**, as seen in Figure 2.5. Remember that UNIX is case sensitive. When the `Password:` prompt appears, type your password on the UNIX system and then press `<Enter>` on the keyboard. Finally, the command line prompt appears on screen, as seen in Figure 2.5.

To terminate your connection type `logout` at the command line prompt and then press `<Enter>` on the keyboard or on a blank line press `<Ctrl+D>`—that is, hold down the `<Ctrl>` and `D` keys on the keyboard at the same time. Logging out is somewhat system dependent, as well as being an operation that can be tailored to a specific installation of UNIX by the local system administrator. In the C shell, the `logout` command is the default way of leaving the system gracefully.

If you use the Bourne shell or Korn shell, holding down `<Ctrl+D>` or typing `exit` will accomplish the same thing. You will then be logged off the system, the current PuTTY session will end, and all PuTTY windows will close.

If you started a new shell during your session and didn’t exit that shell before logging off, UNIX will prompt `Not login shell`, and you will not be able to log off immediately. In this case, press `<Ctrl+D>` and the new shell will terminate. Also, if you started more than one shell and haven’t exited from those shells before you log off, you will have to use `<Ctrl+D>` to terminate each shell individually. On some systems, you type `exit` on the command line to terminate a shell process. In either case, you will then be able to use the logout procedure previously described to leave the system.

2.3.3 Connecting via an SSH Client between UNIX Machines

This way allows a user on one UNIX computer to remote log in and log out of another UNIX computer using the SSH protocol. As detailed in Chapter 11, SSH is an encrypted channel of communication between computers on a LAN or on the Internet.

Before this way can be used, both systems must be able to talk to each other over the SSH channel, which we show how to do in Chapter 11. Also, as previously stated, the user must know a valid username/password pair to be able to log in to the remote system!

We show three possible ways this can happen. First, if the user has already logged into the host successfully from the client before and the authentication keys have not changed. Second, if the user has never logged into the host successfully before from the client. And third, if the user has logged into the host before but the authentication key on the host has changed since the last successful login. These are practical situations one might encounter any time you use this remote login method.

What the user types in is shown in **`bold`** text:

1. Having logged in before successfully:

```
[bob@pcbsd-923] ~% ssh 192.168.0.8
Password for bob@pcbsd-2467: XXX
```

```
Last login: Mon Sep 21 17:20:51 2015 from 192.168.0.13
FreeBSD 10.2-RELEASE-p4 (GENERIC) #0: Tue Aug 18 15:15:36 UTC
    2015
Output truncated...
[bob@pcbsd-2467] ~% Execute command line UNIX commands…
[bob@pcbsd-2467] ~% logout
Connection to 192.168.0.8 closed.
[bob@pcbsd-923] ~%
```

2. Having never logged in before:

```
[bob@pcbsd-923] ~% ssh 192.168.0.6
The authenticity of host '192.168.0.6 (192.168.0.6)' can't be
    established.
RSA key fingerprint is 47:62:a2:9b:24:9e:5e:51:49:3b:80:aa:91:
    a3:fd:de.
No matching host key fingerprint found in DNS.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.0.6' (RSA) to the list of
    known hosts.
Password: XXX
Last login: Mon Sep 21 17:06:59 2015 from 192.168.0.13
Oracle Corporation SunOS 5.11 11.2 June 2014
bob@solaris:~$ Execute command line UNIX commands…
bob@solaris:~$ logout
Connection to 192.168.0.6 closed.
[bob@pcbsd-923] ~%
```

3. Logged in before but host key has changed:

```
[bob@pcbsd-923] ~% ssh 192.168.0.8
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@ WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED! @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-
    middle attack)!
It is also possible that a host key has just been changed.
The fingerprint for the ECDSA key sent by the remote host is
43:e8:cf:33:d5:ed:dd:05:d9:e9:a5:9d:d3:18:1d:2b.
Please contact your system administrator.
Add correct host key in /usr/home/bob/.ssh/known_hosts to get
    rid of this message.
Offending ECDSA key in /usr/home/bob/.ssh/known_hosts:2
ECDSA host key for 192.168.0.8 has changed and you have
    requested strict checking.
Host key verification failed.
[bob@pcbsd-923] ~% cd /usr/home/bob/.ssh
```

```
[bob@pcbsd-923] ~/.ssh% rm known_hosts
[bob@pcbsd-923] ~/.ssh% cd
[bob@pcbsd-923] ~% ssh 192.168.0.8
The authenticity of host '192.168.0.8 (192.168.0.8)' can't be
    established.
ECDSA key fingerprint is 43:e8:cf:33:d5:ed:dd:05:d9:e9:a5:9d:d
    3:18:1d:2b.
No matching host key fingerprint found in DNS.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.0.8' (ECDSA) to the list
    of known hosts.
Password for bob@pcbsd-2467: XXX
Last login: Sat Sep 19 11:24:47 2015 from 192.168.0.13
FreeBSD 10.2-RELEASE-p4 (GENERIC) #0: Tue Aug 18 15:15:36 UTC
    2015
Output truncated...
[bob@pcbsd-2467] ~% Execute command line UNIX commands...
[bob@pcbsd-2467] ~% logout
Connection to 192.168.0.8 closed.
[bob@pcbsd-923] ~%
```

In all of these scenarios, the user is assumed to have an account with the same username, and possibly password, on both client and host systems.

In 2., the keys are generated on host and client after you type in `yes` and press `<Enter>`.

In 3., after the first failed attempt to establish an SSH connection, the error message indicates that the authentication key has changed on the host. Therefore, a removal of the offending key in the file **/usr/home/bob/.ssh/known_hosts** on the client machine **[bob@pcbsd-923]** is done by deleting that file. Then a new key is generated, an exchange can take place, and the login can proceed.

The line in these sessions that reads *Execute command line UNIX commands…* is where the user types in any of the valid UNIX commands we show in this chapter and throughout the rest of this book. Finally, after typing `logout`, the user cuts the SSH channel connection, and is returned to the command line prompt of the local client system.

2.4 FILE MAINTENANCE COMMANDS AND HELP ON UNIX COMMAND USAGE

After your first-time login to a new UNIX system using one of the three ways we described, your first action is to construct and organize your workspace and the files contained in it. The operation of organizing your files according to some logical scheme is known as *file maintenance*. A logical scheme used to organize your files might consist of creating *bins* for storing files according the subject matter of the contents of the files, or according to the dates of their creation. In the following sections, you will type file creation and maintenance commands that produce a structure as shown in Figure 2.6. Complete the operations shown in the following sections in the order they are presented, to get a better overview of what file maintenance really is. Also, it is critical that you review what was

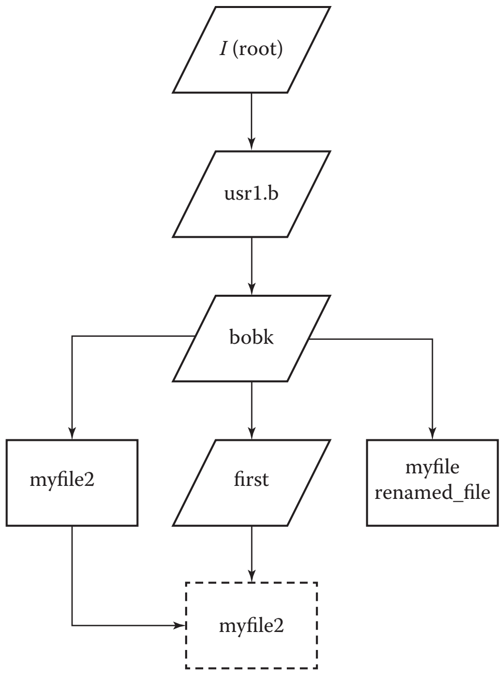


FIGURE 2.6 Example of a file and directory structure.

presented in Section 2.2 regarding the structure of a UNIX command, so that when you begin to type commands for file maintenance, you understand how the syntax of what you are typing conforms to the general syntax of any UNIX command.

2.4.1 File and Directory Structure

When you first log in, you are working in the *home directory*, or folder, of the autonomous user associated with the username and password you used to log in. Whatever directory you are presently in is known as the *current working directory*, and there is only one current working directory active at any given time. It is helpful to visualize the structure of your files and directories using a diagram. Figure 2.6 is an example of a home directory and file structure for a user named **bobk**. In this figure, directories are represented as parallelograms and plain files (e.g., files that contain text or binary instructions) are represented as rectangles. A *pathname*, or path, is simply a textual way of designating the location of a directory or file in the complete file structure of the UNIX system you are working on. For example, the path to the file **myfile2** in Figure 2.6 is **/usr1.b/bobk/myfile2**. The designation of the path begins at the root (/) of the entire file system, descends to the folder named **usr1.b**, and then descends again to the home directory named **bobk**.

As shown in Figure 2.6, the files named **myfile**, **myfile2**, and **renamed_file** are stored under or in the directory **bobk**. Beneath **bobk** is a *subdirectory* named **first**. In the next sections, you will create these files, and the subdirectory structure, in the home directory of the username that you have logged into your UNIX system.

2.4.2 Viewing the Contents of Files

To begin working with files, you create a new file by using the `cat` command. The syntax of the `cat` command is as follows:

SYNTAX

```
cat [options] [file-list]
```

Purpose: Join one or more files sequentially or display them in the console window
Output: Contents of the files in **file-list** displayed on the screen, one file at a time
Commonly used options/features:

`+E`	Display $ at the end of each line
`-n`	Put line numbers on the displayed lines
`-- help`	Display the purpose of the command and a brief explanation of each option

The `cat` command, short for concatenate, allows you to join files. In the example you will join what you type on the keyboard to a new file being created in the current working directory. This is achieved by the redirect character >, which takes what you type at the standard input (in this case the keyboard) and directs it into the file named **myfile**. As stated in Section 2.2, this usage involves the command `cat` but no options, option arguments, or command arguments. It simply uses the command, a redirect character, and a target, or destination, named **myfile**, where the redirection will go.

This is the very simplest example of a *multiple command* typed on the command line, as opposed to a single command, as shown in Section 2.2. In a multiple command, you can string together single UNIX commands in a chain with connecting operators, such as the redirect character shown here.

```
$ cat > myfile
This is an example of how to use the cat command to add plain text
to a file
<Ctrl+D>
$
```

You can type as many lines of text, pressing `<Enter>` on the keyboard to distinguish between lines in the file, as you want. Then, on a new line, when you hold down `<Ctrl+D>`, the file is created in the current working directory, using the command you typed. You can view the contents of this file, since it is a plain text file that was created using the keyboard, by doing the following:

This is a simple example of a single UNIX command.

```
$ more myfile
This is an example of how to use the cat command to add plain text
to a file
$
```

The general syntax of the more command is as follows:

SYNTAX

```
more [options] [file-list]
```

Purpose: Concatenate/display the files in **file-list** on the screen, one screen at a time
Output: Contents of the files in **file-list** displayed on the screen, one page at a time
Commonly used options/features:

`+E/str`	Start two lines before the first line containing `str`
`-nN`	Display N lines per screen/page
`+N`	Start displaying the contents of the file at line number `N`

The `more` command shows one screen of a file at a time. If the file is several pages long, you can proceed to view subsequent pages by pressing the `<Space>` key on the keyboard, or by pressing the `Q` key to quit. Solaris has a command named `pg` that accomplishes the same thing as the `more` command.

2.4.3 Creating, Deleting, and Managing Files

To copy the contents of one file into another file, use the `cp` command. The general syntax of the `cp` command is as follows:

SYNTAX

```
cp [options] file1 file2
```

Purpose: Copy **file1** to **file2**; if **file2** is a directory, make a copy of **file1** in this directory
Output: Copied files
Commonly used options/features:

`-i`	If destination exists, prompt before overwriting
`-p`	Preserve file access modes and modification times on copied files
`-r`	Recursively copy files and subdirectories

For example, to make an exact duplicate of the file named **myfile**, with the new name **myfile2**, type the following:

```
$ cp myfile myfile2
$
```

This usage of the `cp` command has two required command arguments. The first argument is the source file that already exists and which you want to copy. The second argument is the destination file or the name of the file that will be the copy. Be aware that many UNIX commands can take plain, ordinary, or regular files as arguments, or can take directory files as arguments. This can change the basic task accomplished by the command. It is also worth noting that not only can file names be arguments, but pathnames as well. This changes the site or location, in the path structure of the file system, of operation of the command.

In order to change the name of a file or directory, you can use the mv command. The general syntax of the mv command is as follows:

SYNTAX

```
mv [options] file1 file2
mv [options] file-list directory
```

Purpose: First syntax: Rename file1 to file2
Second syntax: Move all the files in file-list to directory
Output: Renamed or relocated files
Commonly used options/features:
`-f` Force the move regardless of the file access modes of the destination file
`-i` Prompt the user before overwriting the destination

In the following usage, the first argument to the mv command is the source file name, and the second argument is the destination name.

```
$ mv myfile2 renamed_file
$
```

It is important at this point to notice the use of spaces in UNIX commands. What if you obtain a file from a Windows 10 system that has one or more spaces in one of the file names? How can you work with this file in UNIX? The answer is simple. Whenever you need to use that file name in a command as an argument, enclose the file name in double quotes ("). For example, you might obtain a file that you have detached from an e-mail message from someone on a Windows 10 system, such as **latest revisions october.txt**.

In order to work with this file on a UNIX system—that is, to use the file name as an argument in a UNIX command—enclose the whole name in double quotes. The correct command to rename that file to something shorter would be:

```
$ mv "latest revisions october.txt" laterevs.txt
$
```

In order to delete a file, you can use the rm command. The general syntax of the rm command is as follows:

SYNTAX

```
rm [options] file-list
```

Purpose: Removes files in **file-list** from the file structure (and disk)
Output: Deleted files
Commonly used options/features:
`-f` Remove regardless of the file access modes of **file-list**
`-i` Prompt the user before removing files in **file-list**
`-r` Recursively remove the files in **file-list** if **file-list** is a directory; use with caution!

To delete the file **renamed_file** from the current working directory, type:

```
$ rm renamed_file
$
```

The most important command you will execute to do file maintenance is the `ls` command. The general syntax for the `ls` command is as follows:

SYNTAX

`ls [options] [pathname-list]`

Purpose: Sends the names of the files and directories in the directory specified by **pathname-list** to the display screen
Output: Names of the files and directories in the directory specified by **pathname-list**, or the names only if **pathname-list** contains file names only
Commonly used options/features:

- **`-F`** Display a slash character (/) after directory names, an asterisk (*) after binary executables, and an "at" character (@) after symbolic links
- **`-a`** Display names of all the files, including hidden files
- **`-i`** Display inode numbers
- **`-l`** Display long list that includes file access modes, link count, owner, group, file size (in bytes), and modification time

The `ls` command will list the names of files or folders in your current working directory or folder. In addition, as with the other commands we have used so far, if you include a complete pathname specification for the `pathname-list` argument to the command, then you can list the names of files and folders along that pathname list. To see the names of the files now in your current working directory, type the following:

```
$ ls
Desktop
Mail
XF86Config.new
kdeinit.core
order.asp.html
order.asp_files
myfile
myfile2$
```

Please note that you will probably not get a listing of the same file names as we did here, because your system will have placed some files automatically in your home directory, as in the example we used, aside from the ones we created together named **myfile** and **myfile2**. Also note that this file name listing does not include the name **renamed_file**, because we deleted that file.

The next command you will execute is actually just an alternate or modified way of executing the `ls` command, one that includes the command name and options. As shown in

Section 2.2, a UNIX command has options that can be typed on the command line along with the command to change the behavior of the basic command. In the case of the `ls` command, the options `l` and `a` produce a longer listing of all ordinary and system (dot) files, as well as providing other attendant information about the files. Don't forget to put the space character between the `s` and the dash. Remember again from Section 2.2 that spaces delimit, or partition, the components of a UNIX command as it is typed on the command line.

Now, type the following command:

```
$ ls -la
drwxr-xr-x  10 bobk  wheel     1024 Oct 11 13:42 .
drwxr-xr-x  17 bobk  wheel      512 Sep 20 16:30 ..
lrwxr-xr-x   1 bobk  wheel       32 Oct 11 13:13
-rw-------   1 bobk  wheel      197 Oct 11 13:13 .ICEauthority
-rw-------   1 bobk  wheel      105 Oct 11 13:13 .Xauthority
-rw-r--r--   2 bobk  wheel      797 Jan 16  2004 .cshrc
-rw-r--r--   2 bobk  wheel      251 Jan 16  2004 .profile
drwxr-xr-x   2 bobk  wheel      512 Apr 15 15:11 .qt
-rwxr-xr-x   1 bobk  wheel       14 Apr 15 08:06 .xinitrc
-rwxr-xr-x   1 bobk  wheel       14 Apr 15 08:06 .xsession
drwx------   3 bobk  wheel      512 Sep 20 16:29 Desktop
drwx------   7 bobk  wheel      512 Apr 15 16:40 Mail
-rw-r--r--   1 bobk  wheel     2798 Apr 17 16:07 XF86Config.new
-rw-------   1 bobk  wheel  7360512 Sep 20 16:29 kdeinit.core
-rw-r--r--   1 bobk  wheel    35394 Apr 17 15:23 order.asp.html
drwxr-xr-x   2 bobk  wheel     1024 Apr 17 15:23 order.asp_files
-rw-r--r--   2 bobk  wheel      797 Jan 16  2004 myfile
-rw-r--r--   2 bobk  wheel      797 Jan 16  2004 myfile2
$
```

As you see in this screen display (which shows the listing of files in our home directory and will not be the same as the listing of files in your home directory), the information about each file in the current working directory is displayed in eight columns. The first column shows the type of file, where `d` stands for directory, `l` stands for symbolic link, and – stands for ordinary or regular file. Also in the first column, the access modes to that file for user, group, and others is shown as `r`, w, or `x`. In the second column, the number of links to that file is displayed. In the third column, the username of the owner of that file is displayed. In the fourth column, the name of the group for that file is displayed. In the fifth column, the number of bytes that the file occupies on disk is displayed. In the sixth column, the date that the file was last modified is displayed. In the seventh column, the time that the file was last modified is displayed. In the eighth and final column, the name of the file is displayed. This way of executing the command is a good way to list more complete information about the file. Examples of using the more complete information are (1) so that you can know the byte size and be able to fit the file on some portable storage medium, or (2) to display the access modes, so that you can alter the access modes to a particular file or directory.

You can also get a file listing for a single file in the current working directory by using another variation of the `ls` command, as follows:

```
$ ls -la myfile
-rw-r--r--  1 bobk  wheel  797 Jan 16 2015 myfile
$
```

This variation shows you a long listing with attendant information for the specific file named **myfile**. A breakdown of what you typed on the command line is 1) **ls**, the command name, 2) **-la** , the options, and 3) **myfile**, the command argument.

What if you make a mistake in your typing and misspell a command name or one of the other parts of a command? Type the following on the command line:

```
$ lx -la myfile
lx: not found
$
```

The `lx:  not  found` reply from UNIX is an error message. There is no `lx` command in the UNIX operating system, so an error message is displayed. If you had typed an option that did not exist, you would also get an error message. If you supplied a file name that was not in the current working directory, you would get an error message, too. This makes an important point about the execution of UNIX commands. If no error message is displayed, then the command executed correctly and the results might or might not appear on screen, depending on what the command actually does. If you get an error message displayed, you must correct the error before UNIX will execute the command as you type it. Typographic mistakes account for about 98% of the errors that beginners make.

2.4.4 Creating, Deleting, and Managing Directories

Another critical aspect of file maintenance is the set of procedures and the related UNIX commands you use to create, delete, and organize directories in your UNIX account on a computer. When moving through the file system, you are either ascending or descending to reach the directory you want to use. The directory directly above the current working directory is referred to as the *parent* of the current working directory. The directory or directories immediately under the current working directory are referred to as the *children* of the current working directory. For more information on file system structure, see Chapter 4. The most common mistake for beginners is misplacing files. They cannot find the file names listed with the `ls` command because they have placed or created the files in a directory either above or below the current working directory in the file structure. When you create a file, if you have also created a logically organized set of directories beneath your own home directory, you will know where to store the file. In the following set of commands, we create a directory beneath the home directory and use that new directory to store a file.

To create a new directory beneath the current working directory, you use the `mkdir` command. The general syntax for the `mkdir` command is as follows:

SYNTAX

```
mkdir [options] dirnames
```

Purpose: Creates directory or directories specified in **dirnames**
Output: New directory or directories
Commonly used options/features:
`-m MODE` Create a directory with given access modes
`-p` Create parent directories that don't exist in the pathnames specified in **dirnames**

To create a child, or subdirectory, named **first** under the current working directory, type the following:

```
$ mkdir first
$
```

This command has now created a new subdirectory named **first** under, or as a child of, the current working directory. Refer back to Figure 2.6 for a graphical description of the directory location of this new subdirectory.

In order to change the current working directory to this new subdirectory, you use the `cd` command. The general syntax for the `cd` command is as follows:

SYNTAX

```
cd [directory]
```

Purpose: Change the current working directory to **directory** or return to the home directory when **directory** is omitted
Output: New current working directory

To change the current working directory to **first** by descending down the path structure to the specified directory named **first**, type the following:

```
$ cd first
$
```

You can always verify what the current working directory is by using the `pwd` command. The general syntax of the `pwd` command is as follows:

SYNTAX

```
pwd
```

Purpose: Displays the current working directory on screen
Output: Pathname of current working directory

You can verify that **first** is now the current working directory by typing the following:

```
$ pwd
/usr1.b/bobk/first
$
```

The output from UNIX on the command line shows the pathname to the current working directory or folder. As previously stated, this path is a textual route through the complete file structure of the computer that UNIX is running on, ending in the current working directory. In the this example of the output, the path starts at **/**, the root of the file system. Then it descends to the directory **usr1.b**, a major branch of the file system on the computer running UNIX. Then it descends to the directory **bobk**, another branch, which is the home directory name for the user. Finally, it descends to the branch named **first**, the current working directory.

On some systems, depending on the default settings, another way of determining what the current working directory is can be done by simply looking at the command line prompt. This prompt may be prefaced with the complete path to the current working directory, ending in the current working directory.

You can ascend back up to the home directory, or the parent of the subdirectory **first**, by typing the following:

```
$ cd
$
```

An alternate way of doing this is to type the following, where the tilde character (~) resolves to, or is a substitute for, the specification of the complete path to the home directory:

```
$ cd ~
$
```

To verify that you have now ascended up to the home directory, type the following:

```
$ pwd
/usr1.b/bobk
$
```

You can also ascend to a directory above your home directory, sometimes called the parent of your current working directory, by typing the following:

```
$ cd ..
$
```

In this command, the two periods (`..`), represent the parent, or branch above the current working directory. Don't forget to type a space character between the `d` and the first period. To verify that you have ascended to the parent of your home directory, type the following:

```
$ pwd
/usr1.b
$
```

To descend to your home directory, type the following:

```
$ cd
$
```

To verify that there are two files in the home directory that begins with the letters `my`, type the following command:

```
$ ls my*
myfile  myfile2
$
```

The asterisk following the `y` on the command line is known as a *metacharacter*, or a character that represents a pattern; in this case, the pattern is any set of characters. When UNIX interprets the command after you press the `<Enter>` key on the keyboard, it searches for all files in the current working directory that begin with the letters `my` and end in anything else.

Another aspect of organizing your directories is movement of files between directories, or changing the location of files in your directories. For example, you now have the file **myfile2** in your home directory, but you would like to move it into the subdirectory named **first**. See Figure 2.6 for a graphic description to change the organization of your files at this point. To accomplish this, you can use the second syntax method illustrated for the `mv file-list directory` command to move the file **myfile2** down into the subdirectory named **first**. To achieve this, type the following:

```
$ mv myfile2 first
$
```

To verify that **myfile2** is indeed in the subdirectory named first, type the following:

```
$ cd first
$ ls
myfile2
$
```

You will now ascend to the home directory, and attempt to remove or delete a file with the `rm` command. Caution: you should be very careful when using this command, because once a file has been deleted, the only way to recover it is from archival backups that you or the system administrator have made of the file system.

```
$ cd
$ rm myfile2
rm: myfile2: No such file or directory
$
```

You get the error message because in the home directory, the file named **myfile2** does not exist. It was moved down into the subdirectory named first.

Directory organization also includes the ability to delete empty or nonempty directories. The command that accomplishes the removal of empty directories is `rmdir`. The general syntax of the `rmdir` command is as follows:

SYNTAX

`rmdir [options] dirnames`

Purpose: Removes the empty directories specified in **dirnames**
Output: Removes directories
Commonly used options/features:
`-p` Remove empty parent directories as well
`-r` Recursively delete files and subdirectories beneath the current directory

To delete an entire directory below the current working directory, type the following:

```
$ rmdir first
rmdir: first: Directory not empty
$
```

Since the file **myfile2** is still in the subdirectory named **first**, **first** is not an empty directory, and you get the error message that the `rmdir` command will not delete the directory. If the directory was empty, `rmdir` would have accomplished the deletion. One way to delete a nonempty directory is by using the `rm` command with the `-r` option. The `-r` option recursively descends down into the subdirectory and deletes any files in it before actually deleting the directory itself. Be cautious with this command, since you may inadvertently delete directories and files with it. To see how this command deletes a nonempty directory, type the following:

```
$ rm -r first
$
```

The directory **first** and the file **myfile2** are now removed from the file structure.

2.4.5 Obtaining Help with the Man Command

A very convenient utility available on UNIX systems is the online help feature, achieved via the use of the `man` command. The general syntax of the `man` command is as follows:

SYNTAX

```
man [options][-s section] command-list
man -k keyword-list
```

Purpose: First syntax: Display UNIX Reference Manual pages for commands in **command-list** one screen at a time
Second syntax: Display summaries of commands related to keywords in **keyword-list**
Output: Manual pages one screen at a time
Commonly used options/features:

`-k keyword-list`	Search for summaries of keywords in **keyword-list** in a database and display them
`-s sec-num`	Search section number **sec-num** for manual pages and display them

To get help by using the `man` command, on usage and options of the `ls` command, for example, type the following (shown for PC-BSD):

```
$ man ls

LS(1))                   FreeBSD General Commands Manual LS(1)
NAME
    ls - list directory contents
SYNOPSIS
    ls [-ABCFGHLPRTWZabcdfghiklmnopqrstuwx1] [file ...]
DESCRIPTION
    For each operand that names a file of a type other than
directory, ls displays its name as well as any requested,
associated information. For each operand that names a file of type
directory, ls displays the names of files contained within that
directory, as well as any requested, associated information.
If no operands are given, the contents of the current directory
are displayed. If more than one operand is given, non-directory
operands are displayed first; directory and non-directory operands
are sorted separately and in lexicographical order.
    The following options are available:
Press <SPACE> to continue, or q to quit q
$
```

This output from UNIX is a UNIX *manual page*, or *man page*, which gives a synopsis of the command usage showing the options, and a brief description that helps you understand how the command should be used. Typing `q` after one page has been displayed, as seen in the example, returns you to the command line prompt. Pressing the space key on the keyboard would have shown you more of the content of the manual pages, one screen at a time, related to the `ls` command.

To get help in using all the UNIX commands and their options, use the `man man` command to go to the UNIX reference manual pages.

TABLE 2.1 Sections of the UNIX Manual

Section	What It Describes
1	User commands
2	System calls
3	Language library calls (C, FORTRAN, etc.)
4	Devices and network interfaces
5	File formats
6	Games and demonstrations
7	Environments, tables, and macros for troff
8	System maintenance–related commands

The pages themselves are organized into eight sections, depending on the topic described and the topics that are applicable to the particular system. Table 2.1 lists the sections of the manual and what they contain. Most users find the pages they need in Section 2.1. Software developers mostly use library and system calls and thus find the pages they need in Sections 2.2 and 2.3. Users who work on document preparation get the most help from Section 2.7. Administrators mostly need to refer to pages in Sections 2.1, 2.4, 2.5, and 2.8.

The manual pages comprise multipage, specially formatted, descriptive documentation for every command, system call, and library call in UNIX. This format consists of seven general parts: name, synopsis, description, list of files, related information, errors, warnings, and known bugs. You can use the `man` command to view the manual page for a command. Because of the name of this command, the manual pages are normally referred to as UNIX man pages. When you display a manual page on the screen, the top-left corner of the page has the command name with the section it belongs to in parentheses, as with `LS(1)`, seen at the top of the output manual page.

The command used to display the manual page for the `passwd` command is:

```
$ man passwd
```

The manual page for the `passwd` command now appears on the screen, but we do not show its output. Because they are multipage text documents, the manual pages for each topic take up more than one screen of text to display their entire contents. To see one screen of the manual page at a time, press the space bar on the keyboard. To quit viewing the manual page, press the `Q` key on the keyboard.

There is no `-k` option listed on a PC-BSD system, but there is one on a Solaris system. And the `-k` option works on both systems!

Now type this command:

```
$ man pwd
```

If more than one section of the man pages has information on the same word and you are interested in the man page for a particular section, you can use the `-S` option (in Solaris it is lowercase s). The following command line therefore displays the man page for the read system call and not the man page for the shell command read.

```
$ man -S2 read
```

The command `man -S3 fopen fread strcmp` sequentially displays man pages for three C library calls: **fopen**, **fread**, and **strcmp**.

On a Solaris system, using the `man` command includes typing the command with the `-k` option, thereby specifying a keyword that limits the search. The search then yields useful man page headers from all the man pages on the system that contain just the keyword reference. For example, the following session yields the on-screen output on a Solaris system:

```
% man -k passwd
1. passwd(4) /usr/share/man/man4/passwd.4
passwd - password file
2. passwd(1openssl) /usr/share/man/man1openssl/passwd.1openssl
passwd - compute password hashes
3. passwd(1) /usr/share/man/man1/passwd.1
passwd - change login password and password attributes
4. slapd-passwd(5oldap) /usr/share/man/man5oldap/
slapd-passwd.5oldap
slapd-passwd - /etc/passwd backend to slapd
5. getpw(3c) /usr/share/man/man3c/getpw.3c
getpw - get passwd entry from UID
6. vino-passwd(1) /usr/share/man/man1/vino-passwd.1
vino-passwd - change vino login password
7. pwconv(1m) /usr/share/man/man1m/pwconv.1m
pwconv - installs and updates /etc/shadow with information from /
etc/passwd
8. SSL_CTX_set_default_passwd_cb(3openssl) /usr/share/man/
man3openssl/SSL_CTX_set_default_passwd_cb.3openssl
SSL_CTX_set_default_passwd_cb,
SSL_CTX_set_default_passwd_cb_userdata
- set passwd callback for encrypted PEM file handling
9. SSL_CTX_set_default_passwd_cb_userdata(3openssl) /usr/share/
man/man3openssl/SSL_CTX_set_default_passwd_cb_userdata.3openssl
SSL_CTX_set_default_passwd_cb,
SSL_CTX_set_default_passwd_cb_userdata
- set passwd callback for encrypted PEM file handling
```

2.4.6 Other Methods of Obtaining Help

To get a short description of what any particular UNIX command does, you can use the `whatis` command. This is similar to the command `man -f`. The general syntax of the `whatis` command is as follows:

SYNTAX

`whatis keywords`

Purpose: Search the whatis database for abbreviated descriptions of each keyword
Output: Prints a one-line description of each keyword to the screen

The following is an illustration of how to use `whatis`.
The output of the two commands are truncated.

```
$ whatis man
…
man(1)      -format and display the online manual pages
…
$
```

You can also obtain short descriptions of more than one command by entering multiple arguments to the `whatis` command on the same command line, with spaces between each argument. The following is an illustration of this method:

```
$ whatis login set setenv
…
login(1)    -sign on
…
set(1)      -set runtime parameters for session
…
setenv(1)   -change or add an environment variable
…
$
```

The following in-chapter exercises ask you to use the `man` and `whatis` commands to find information about the `passwd` command. After completing the exercises, you can use what you have learned to change your login password on the UNIX system that you use.

EXERCISE 2.1

Use the `man` command with the `-k` option (in both PC-BSD and Solaris) to display abbreviated help on the `passwd` command. Doing so will give you a screen display similar to that obtained with the `whatis` command, but it will show all apropos command names that contain the characters `passwd`.

EXERCISE 2.2

Use the `whatis` command (in both PC-BSD and Solaris) to get a brief description of the `passwd` command shown in Exercise 2.1, and then note the difference between the commands `whatis passwd` and `man -k passwd`.

2.5 UTILITY COMMANDS

There are several major commands that allow the beginner to be more productive when using the UNIX system. A sampling of these kinds of utility commands is given in the following sections, and is organized as system setups, general utilities, and communications commands.

2.5.1 Examining System Setups

The `whereis` command allows you to search along certain prescribed paths to locate utility programs and commands, such as shell programs. The general syntax of the `whereis` command is as follows:

SYNTAX

```
whereis [options] filename
```

Purpose: Locate the binary, source, and man page files for a command
Output: The supplied names are first stripped of leading pathname components and extensions, then pathnames are displayed on screen
Commonly used options/features:

-b	Search only for binaries
-s	Search only for source code

For example, if you type the command `whereis csh` on the command line, you will see a list of the paths to the C shell program files themselves. Note that the paths to a built-in, or internal, command cannot be found with the `whereis` command. We provide more information about internal and external shell commands in Chapter 10.

When you first log on, it is useful to be able to view a display of information about your `userid`, the computer or system you have logged on to, and the operating system on that computer. These tasks can be accomplished with the `whoami` command, which displays your `userid` on the screen. The general syntax of the `whoami` command is as follows:

SYNTAX

```
whoami
```

Purpose: Displays the effective user id
Output: Displays your effective user id as a name on standard

The following shows how our system responded to this command when we typed it on the command line.

```
$ whoami
bobk
$
```

The following in-chapter exercises give you the chance to use `whereis`, `whoami`, and two other important utility commands, `who` and `hostname` to obtain important information about your system.

EXERCISE 2.3

On a PC-BSD system, use the `whereis` command to locate binary files for the Korn shell, the Bourne shell, the Bourne Again shell, the C shell, and the Z shell. Are any of these shell programs not available on your system?

EXERCISE 2.4

Use the `whoami` command to find your username on the system that you're using. Then use the `who` command to see how your username is listed, along with other users of the same system. What is the on-screen format of each user's listing that you obtained with the `who` command? Try to identify the information in each field on the same line as your username.

EXERCISE 2.5

Use the `hostname` command to find out what host computer you are logged on to. Can you determine from this list whether you are using a stand-alone computer or a networked computer system? Explain how you can know the difference from the list that the `hostname` command gives you.

2.5.2 Printing and General Utility Commands

2.5.2.1 For PC-BSD

A very useful and common task performed by every user of a computer system is the printing of text files at a printer. The command to perform printing on a PC-BSD system is `lpr`. The general syntax of the `lpr` command is as follows:

SYNTAX

```
lpr [options] filename
```

Purpose: Send files to the printer
Output: Files sent to the printer queue as print jobs
Commonly used options/features:

`-P printer`	Send output to the named printer
`-# copies`	Produce the number of copies indicated for each named file

The following `lpr` command, when using PC-BSD, accomplishes the printing of the file named **order.eps** at the printer designated on our system as **spr**. Remember from Section 2.2 that no space is necessary between the option (in this case `-P`) and the option argument (in this case `spr`).

```
$ lpr -Pspr order.eps
$
```

The following `lpr` command, when using PC-BSD, accomplishes the printing of the file named **memo1** at the default printer.

```
$ lpr memo1
$
```

The following multiple command combines the `man` command and the `lpr` command, and ties them together with the UNIX *pipe* (|) redirection character, to print the man pages describing the `ls` command at the printer named `hp1`. This will work when using PC-BSD.

```
$ man ls | lpr -Php1
$
```

2.5.2.2 For Solaris

The following shows how to perform printing tasks on Solaris using the `lp` command.

The general syntax of the `lp` command for Solaris is as follows:

SYNTAX

```
lp [options][option arguments] file(s)
```

Purpose: Submit files for printing on a designated system printer, or alter pending print jobs

Output: Printed files or altered print queue

Commonly used options/features:

`-d destination`	Print to the specified destination
`-P pagelist`	Print selected pages as specified in **pagelist**

In the first command, the file to be printed is named **file1**. In the second command, the files to be printed are named **sample** and **phones**. Note that the `-d` option is used to specify which printer to use. The option to specify the number of copies is `-n` for the `lp` command.

```
$ lp -d spr file1
request id is spr-983 (1 file(s))
$ lp -d spr -n 3 sample phones
request id is spr-984 (2 file(s))
$
```

Among the most useful of the general purpose, personal productivity utility commands, the `cal` command displays a calendar for a year or a month. The general syntax of the `cal` command is as follows:

SYNTAX

```
cal [[month]year]
```

Purpose: Displays calendar on screen as text

Output: Displays a calendar of the month or year

The optional parameter `month` can be between 1 and 12, and `year` can be 0–9999. Just like the UNIX system, the `cal` command is Y2K compliant. If no argument is specified, the command displays the calendar for the current month of the current year. If only one parameter is specified, it is taken as the year. Thus the `cal 3 2005` command displays the calendar for March 2005. The command `cal 1969` displays the calendar for the year 1969, the year the UNIX operating system was born.

2.5.3 Communications Commands

The `write` command is used to send a message to another user who is currently logged on to the system. The syntax and a brief description of the command is as follows:

SYNTAX

```
write username [terminal]
```

Purpose: Write on the terminal screen or console window of the user with login name username; the user must be logged on to the system, and the user's terminal must have write access privilege given by the `mesg` command.
Output: Message on another user's console window.

The example shown in the following command line dialog session illustrates the use of this command. The prerequisite for executing the `write` command is execution of the `mesg y` command by both sender (in anticipation of a reply) and receiver to allow writing to their respective terminal screens or console windows. The `who` command is used to determine whether the person to whom you want to write is logged on. In this case, both sender (**sarwar**) and receiver (**bobk**) are logged on to the computer **upibm7**, **sarwar** at terminal **ttyp0** and **bobk** at terminal **ttyC2**. The receiver's screen is garbled with the message, but no harm is caused to any work that the user is doing. Under the shell, pressing `<Enter>` performs the trick of resetting the screen, and inside the `vi` editor (discussed in Chapter 3), the screen can be reset by pressing the `<Ctrl>` and `R` keys on the keyboard at the same time. Notice also that the sending of the message is accomplished by holding down the `<Ctrl>` and `D` keys on the keyboard at the same time.

```
Sender's (sarwar) screen
$mesg y
$who
bobk       upibm7:ttyC2     Oct12     13:47 :34
sarwar     upibm7:ttyp0     Oct12     14:20 :15
$write bobk ttyC2
Bob,
How are the new chapter revisions coming along?
Take care,
Mansoor
<Ctrl+D>
Receiver's (bobk) screen
```

```
$mesg y
$
Message from sarwar@upibm7.egr.up.edu on ttyp0 at 14:26
Bob,
    How are the new chapter revisions coming along?
Take care,
Mansoor
EOF
```

The `mesg` command enables or disables real-time one-way messages and chat requests from other users with the write and talk commands, respectively. The `mesg y` command permits others to initiate communication with you by using the `write` or `talk` command. If you think that you are bothered too often with `write` or `talk`, you can turn off the permission by executing the `mesg n` command. When you do so, a user who runs a `write` or `talk` command sees the message `Permission denied`. When the `mesg` command is used without an argument, it returns the current value of permission, `n` or `y`.

The `biff` command lets the system know whether you want to be notified immediately of an incoming e-mail message. The system notifies you by sounding a beep on your terminal. You can use the command `biff y` to enable notification and `biff n` to disable notification. When the `biff` command is used without an argument, it displays the current setting, `n` or `y`.

2.6 COMMAND ALIASES

The `alias` command can be used to create pseudonyms, or nicknames, for commands. The `alias` command has one syntax in the Bourne, Korn, and Bourne Again (Bash; the default shell in Solaris) shells, and another in the C shell (the default shell for PC-BSD); both forms are illustrated in the following example. The general syntax for the `alias` command is as follows:

SYNTAX

```
alias [name [=string] ...]in Bourne, Korn, Bash shells
alias [name [string]]in C shell
```

Purpose: Create pseudonym `string` for the command `name`
Output: Pseudonyms that can be used for commands

Nicknames are usually created for commands, but they can also be used for other items, such as naming e-mail groups. Both C shell and Bash allow you to create aliases from the command line one at a time, or put them multiply in the resource file for the particular shell.

Command aliases can be placed in the **.profile** file or the **.login** file, but they are typically placed in the **.bashrc** file (for the Bash shell in Solaris) and the **.cshrc** file (for the C

TABLE 2.2 Some Useful Aliases for Various Shells

Bourne, Korn, and Bash Shells	C Shell
`alias dir='ls -la\!*'`	`alias dir 'ls -la\!*'`
`alias rename='mv\!*'`	`alias rename 'mv\!*'`
`alias spr='lpr -Pspr\!*'`	`alias spr 'lpr -Pspr\!*'`
`alias ls='ls -C'`	`alias ls 'ls -C'`
`alias ll='ls -ltr'`	`alias ll 'ls -ltr'`
`alias page='more'`	`alias page 'more'`

shell in PC-BSD). The **.profile** or **.login** file executes when you log on, and the **.cshrc** or **.bashrc** file executes every time you start a C or Bourne shell.

Table 2.2 lists some useful aliases to put in one of these files. If set in your environment by any of the these means, the aliases in the session below allow you to use the names `dir`, `rename`, `spr`, `ls`, `ll`, and `page` as commands, substituting them for the actual commands given in quotes. Thus when you type `dir unixbook`, the shell executes the `ls -la unixbook` command.

When you use the `alias` command without any argument, it lists all the aliases currently set by default.

The following session illustrates the use of this command with a Bourne, Korn, or Bash shell.

The aliases shown are those found on our PC-BSD system, and may not be the same as the ones defined by default on your system.

```
$ alias
dir='ls -la'
rename='mv'
spr='lpr -Pspr'
ls='ls -C'
ll='ls -ltr'
page='more'
$
```

Running the same command with the C shell produces the following output:

```
% alias
dir     ls -la
rename       mv
spr     lpr -Pspr
ls      ls -C
ll      ls -ltr
page    more
%
```

You can use the `unalias` command to remove one or more aliases from the alias list.

In Solaris, while in the Bash shell, you can use the `unalias -a` option to remove all aliases from the alias list. You can also use `unalias -a` in PC-BSD if you are in the Bash shell. You cannot use `unalias -a` in the C shell by default in PC-BSD or Solaris, you must `unalias` each alias one at a time.

In the following PC-BSD/Solaris/Bash session, the first of the two `unalias` commands removes the `alias` for `ls`, and the second removes all of the aliases from the alias list. Note that the output of the first `alias` command does not contain an alias for the `ls` command after the `unalias ls` command has been executed. Use of the second `alias` command produces no output because the `unalias -a` command removes all the aliases from the alias list.

```
$ unalias ls
$ alias
dir='ls -la'
rename='mv'
spr='lpr -Pspr'
ll='ls -ltr'
page='more'
$ unalias -a
$ alias
$
```

In the following in-chapter exercises, you will use the `write`, `alias`, and `unalias` commands to practice their syntax and gain more insight into their utility. You will also examine a system file that keeps track of users that can log in.

EXERCISE 2.6

Use the `write` command to communicate with a friend who is logged on to the system.

EXERCISE 2.7

Use the `alias` command to display the nicknames (aliases) of commands in your system, if there are any. If there aren't any, create a few useful ones for yourself according to what you might use frequently and beneficially as a nicknamed command. Then, in PC-BSD use `unalias` to remove one or more of them. In Solaris use `unalias -a` to remove all of the aliases. After you have unaliased all the defaults or defined aliases, how do you reinstate them?

EXERCISE 2.8

Display the contents of the **/etc/passwd** file on your system to determine how many users can log on to the system.

Table 2.3 shows some useful commands for beginners.

TABLE 2.3 Useful Commands for the Beginner

Command	What It Does
`<Ctrl+D>`	Terminates a process or command
`alias`	Allows you to create pseudonyms for commands
`biff`	Notifies you of new e-mail
`cal`	Displays a calendar on screen
`cat`	Allows joining of files
`cd`	Allows you to change the current working directory
`cp`	Allows you to copy files
`exit`	Ends a shell that you have started
`hostname`	Displays the name of the host computer that you are logged on to
`login`	Allows you to log on to the computer with a valid username/password pair
`lpr or lp`	Allows printing of text files
`ls`	Allows you to display names of files and directories in the current working directory
`man`	Allows you to view a manual page for a command or topic
`mesg`	Allows or disallows writing messages to the screen
`mkdir`	Allows you to create a new directory
`more`	Allows viewing of the contents of a file one screen at a time
`mv`	Allows you to move the path location of, or rename, files
`passwd`	Allows you to change your password on the computer
`pg`	Solaris command that displays one screen of a file at a time
`pwd`	Allows you to see the name of the current working directory
`rm`	Allows you to delete a file from the file structure
`rmdir`	Allows deletion of directories
`talk`	Allows you to send real-time messages to other users
`telnet`	Allows you to log on to a computer on a network or the Internet
`unalias`	Allows you to undefine pseudonyms for commands
`uname`	Displays information about the operating system running the computer
`whatis`	Allows you to view a brief description of a command
`whereis`	Displays the path(s) to commands and utilities in certain key directories
`who`	Allows you to find out login names of users currently on the system
`whoami`	Displays your username
`write`	Allows real-time messaging between users on the system

2.7 INTRODUCTION TO UNIX SHELLS

When you log on and enter a CUI using a console window or terminal, the UNIX system starts running a program that acts as an interface between you and the UNIX kernel. This program, called a *UNIX shell*, executes the commands that you have typed on the keyboard. When a shell starts running, it gives you a prompt and waits for your commands. When you type a command and press `<Enter>`, the shell interprets your command and executes it. If you type a nonexistent command, the shell tells you this, then redisplays the prompt and waits for you to type the next command. Because the primary purpose of the shell is to interpret your commands, it is also known as the *UNIX command line interpreter.*

A shell command can be internal/built-in or external. The code to execute an internal command is part of the shell process, but the code to process an external command resides in a file in the form of a binary executable program file or a shell script. (We describe in detail how a shell executes commands in Chapter 10.) Because the shell executes commands entered from the keyboard, it terminates when it finds out that it cannot read anything else from the keyboard. You can inform your shell of this by pressing `<Ctrl+D>` at the beginning of a new line. As soon as the shell receives `<Ctrl+D>`, it terminates and logs you off the system. The system then displays the `login:` prompt again, informing you that you need to log on again in order to use it.

The shell interprets single UNIX commands that are structured according to Section 2.2—that is, by assuming that the first word in a command line is the name of the command that you want to execute. It assumes that any of the remaining words starting with a hyphen (-) are options (possibly followed by option arguments) and that the rest are the command arguments.

After reading your command line, it determines whether the command is an internal or external command. It processes all internal commands by using the corresponding code segments that are within its own code. To execute an external command, it searches several directories in the file system structure (see Chapter 4), looking for a file that has the name of the command. It then assumes that the file contains the code to be executed and runs the code.

The names of the directories that a shell searches to find the file corresponding to an external command are stored in the shell variable named `PATH` (or `path` in the C shell). Directory names are separated by colons in the Bourne, Korn, and Bash shells and by spaces in the C shell. The directory names stored in the PATH variable form what is known as the *search path* for the shell. You can view the search path for your variable by using the `echo $PATH` command in the Bourne, Korn, Bash, and C shells.

The following are two sample sessions run with this command in the login shell, first the Bourne shell and then second the C shell. Note that in the Bourne shell the search path contains the directory names separated by colons and that in the C shell the directory names are separated by spaces.

```
$ echo $PATH
/usr/sbin:/usr/X11/include/X11:.:/users/faculty/sarwar/bin:/usr/
ucb
:/bin:/usr/bin:/usr/include:/usr/X11/lib:/usr/lib:/etc:/usr/etc:/
usr
/local/bin:/usr/local/lib:/usr/local/games:/usr/X11/bin
$
% echo $path
/usr/sbin /usr/X11/include/X11 . /users/faculty/sarwar/bin /usr/
ucb /bin
/usr/bin /usr/include /usr/X11/lib /usr/lib /etc /usr/etc /usr/
local/bin /usr/local/lib /usr/local/games /usr/X11/bin
%
```

The PATH (or path) variable is defined in a hidden file (also known as a dot file) called **.profile** (Solaris) or **.login** (PC-BSD). If you can't find this variable in one of those files, it is in the start-up file (also a dot file) specific to the shell that you're using. You can change the search path for your shell by changing the value of this variable. To change the search path temporarily for your current session only, you can change the value of PATH at the command line. For a permanent change, you need to change the value of this variable in the corresponding dot file.

In the following Bash shell example, the search path was augmented by two directories, ~/**bin** and **.** (current directory). Moreover, the search starts with ~/**bin** and ends with the current directory.

Be careful when editing or changing the PATH variable, so that you don't lose any component of the default search path set by the system administrator for all users of the system.

```
$ PATH=~/bin:$PATH:.
$
```

You can determine your login shell by using the `echo $SHELL` command, as described in Section 2.8.3. Each shell has several other environment variables set up in a hidden file associated with it. We describe these files in Section 2.8.4 and present a detailed discussion of UNIX files in Chapter 4.

2.8 VARIOUS UNIX SHELLS

Every UNIX system comes with a variety of shells, with the Bash and C shells (the default shells in our base systems, Solaris and PC-BSD) being the most common. The Bourne, Korn, TC, and Z shells are less popular, but offer advantages for certain applications and ways of working with the UNIX system. When you log on, one particular type of shell starts execution. This shell is known as your *login shell*, and it is determined by the system administrator of your UNIX system. If you want to use a different shell, you can do so by running a corresponding command available on your system. For example, if your login shell is Bash, but you want to use the C shell, you can do so by using the `csh` command.

2.8.1 Shell Programs

Essentially the shell program itself, which is implemented in the C programming language, allows you to do interpreted programming (as opposed to compiled programming). It does this in two senses: firstly, so you can employ simple, single or complex, multiple UNIX commands connected by redirection operators and/or utilities such as `sed`, `awk` or `grep`, to do common tasks, and secondly via user-written script files, coded in the shell interpreted language, that automate and simplify those common, perhaps highly repetitive, tasks. This interpreted language has all the features of any other structured, high-level programming language, such as Perl, Tcl, or Python. The shell language is just not as complex as the other common scripting languages. This fact should tell you why there are so many different shells, just as there are many different high-level programming and scripting languages.

TABLE 2.4 Shell Locations and Program Names

Shell	Location on PC-BSD System	Program (Command) Name
rc	NA	`rc`
Bourne shell	/bin/sh	`sh`
C shell	/bin/csh	`csh`
Bourne Again shell	/usr/local/bin/bash	`bash`
Z shell	NA	`zsh`
Korn shell	NA	`ksh`
TC shell	/bin/tcsh	`tcsh`

Programming languages have a tendency to evolve and grow with time, depending on the needs of users, and shell programs are typical of this evolution. Table 2.4 contains a list of the most common shells, their location on a PC-BSD system, and the program names of those shells.

The locations shown in Table 2.4 are typical for most UNIX systems. Consult your instructor or system administrator if you can't find the location shown for a shell on your system or if you can't use the `whereis` command, as shown in Section 2.5.1.

Figure 2.7 traces the development of various shell families, and indicates the increasing functionality of each family as it appears higher in the hierarchy. The Bourne shell (`sh`) is the *grandmother* of the main shell families and has nearly the least level of functionality. Near the top of the hierarchy is the Korn shell (`ksh`), which includes all the functionality of the Bourne shell and much more. The `rc` and `zsh` shells are outliers that cannot be readily associated with any of the primary shell families.

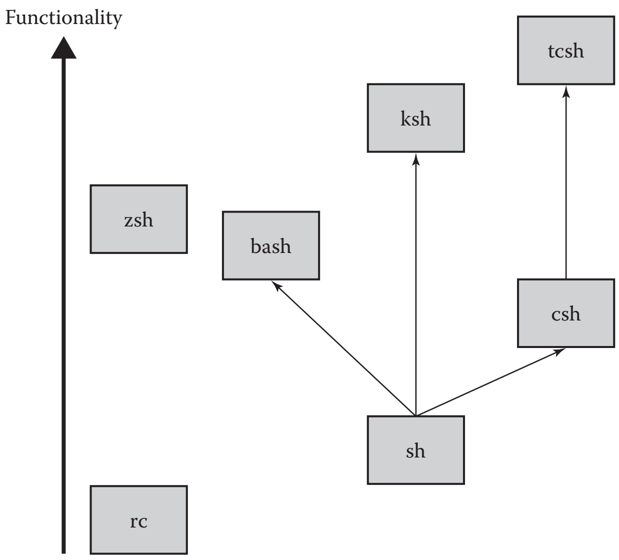


FIGURE 2.7 Shell families and their relative functionalities.

2.8.2 Which Shell Suits Your Needs?

Most shells perform similar functions, and knowing the details of how they do so is important in deciding which shell to use for a particular task. Also, using more than one shell during a session is a common practice, especially among shell script file programmers. For example, you might use the Bourne or Korn shell for their programming capabilities and use the C shell to execute individual commands. We discuss this example further in Section 2.8.3. The similarities of major shell functions are summarized in Table 2.5.

2.8.3 Ways to Change Your Shell

You can easily determine what your default shell is by typing `echo $SHELL` on the command line when you first log on to your computer system.

The question is: Why would you want to change your default shell, or for that matter, even use an additional shell? The answer is that you want the greater, or in some sense qualitatively different, functionality of another shell.

For example, your default shell might be the C shell (`csh`). A friend of yours offers you a neat and useful Bourne shell script that allows you to take advantage of the Bourne shell script programming capabilities, a script that wouldn't work if it ran under the C shell. You can use this script by running the Bourne shell at the same time you are running the default C shell. Because UNIX is a multiprocess operating system, more than one command line interpreter at a time can be active. That doesn't mean that a single command will be interpreted multiply; it simply means that input, output, and errors are "hooked" into whatever shell process has control over them currently. (See Chapter 10 for more information about process and shell command input/output.)

You can change your shell in one of two ways:

1. Changing to a new default for every subsequent login session on your system, and
2. Creating additional shell sessions running on top of, or concurrently with, the default shell.

The premise of both methods is that the shell you want to change to is available on your system.

To change your default shell, after you have logged on, type `chsh` and then press `<Enter>`. Depending on your system, you will be prompted for the name of the shell you want to change to.

On a PC-BSD system, you are prompted for the superuser password in order to accomplish the `chsh` command. There is no `chsh` command available in Solaris.

TABLE 2.5 Shell Similarities

Function	Description
Execution	The ability to execute programs and commands
I/O handling	The control of program and command input and output
Programming	The ability to execute sequences of programs and commands

Type the location of the shell you want to change to—for example, **/usr/bin/sh** to change to the Bourne shell. If this method doesn’t work on your system, consult your instructor or system administrator for more help.

To create or run additional shells on top of your default shell, simply type the name of the shell program (see Table 2.4) on the command line whenever you want to run that shell. The following session illustrates the use of this method to change a default Bash shell, which uses the `$` as the shell prompt, to a C shell, which shows the `%` as the shell prompt.

```
$ echo $SHELL
/usr/bin/bash
$ csh
%
```

The first command line allows you to determine your default shell. In this case, the system shows you that the default setting is the Bash shell. The second command line allows you to run the C shell. The fourth line shows that you have been successful, because the default C shell prompt appears on your display. If the C shell was not available on your system or was inaccessible to you, you would get an error message after the third line. If your search path does not include `/usr/bin`, you either have to type `/usr/bin/csh` in place of `csh`, or include `/usr/bin` in your shell’s search path and then use the `csh` command.

To terminate or leave this new, temporary shell and return to your default login shell, hold down `<Ctrl+D>` on a blank line. If this way of terminating the new shell doesn’t work, type `exit` on the command line and then press `<Enter>`. By doing so, you halt the running of the new shell, and the default shell prompt appears on your display. If you have opened a console or terminal window on your desktop, typing `exit` also closes this console or terminal window.

The following in-chapter exercises ask you to determine whether various shells are available on your system by using the `whereis` command and, for those that are available, to read the manual pages for them by using the `man` command.

EXERCISE 2.9

Using the `whereis` command illustrated in Section 2.5.1, verify the locations of the various shells listed in Table 2.4. Are all these shells available on your system? Where are they located if you do not find them at the locations shown in Table 2.4?

EXERCISE 2.10

Using the `man` command illustrated in 2.4.5, read the manual pages for each shell listed in Table 2.4 that is on your system.

2.8.4 Shell Start-Up Files and Environment Variables

The actions of each shell, the mechanics of how it executes commands and programs, how it handles the command and program I/O, and how it is programmed, are affected by the setting of certain *environment variables*.

Each UNIX system has an initial system start-up file, usually named **.profile** in Solaris and **.login** in PC-BSD. This file contains the initial settings of important environment variables for the shell and some other utilities. In addition, hidden files for specific shells are executed when you start a particular shell. Known as the shell start-up files, they are **.cshrc** for C shell and **.bashrc** for Bash. These hidden files are initially configured by the system administrator for secure use by all users. Table 2.6 lists some important environment variables common to Bash, Bourne, Korn, and C shells; the C shell variable name, where applicable, is in lowercase following the Bash, Bourne, and Korn shell variable name. Note that your system administrator may not have set some of these variables, such as `ENV`.

The following in-chapter exercises let you view the settings of your environment variables. They assume that you are initially running the Bourne or Korn shells. If you aren't, run either of those shells as described in Section 2.8.3 and then do the exercises.

EXERCISE 2.11

At the default login shell prompt for your system, type `set | more` and then press `<Enter>`. What is displayed on your screen? Identify and list the settings for all the environment variables shown in Table 2.6.

EXERCISE 2.12

At the shell prompt, type `csh` or `bash` depending on your default system login shell, and then press `<Enter>`. Next, type `setenv | more` and then press `<Enter>`. Identify and list the settings for all the environment variables shown in Table 2.6.

In addition to the shells, several other programs have their own hidden files. These files are used to set up and configure the operating environment within which these programs execute. We discuss some of these hidden files in Chapters 4 and 5. They are called hidden files because when you list the names of files contained in your home directory—for example, with the `ls -l` command and option (see Chapter 4)—these files do not appear on the list. The hidden file names always start with a period (.), such as **.login**.

TABLE 2.6 Shell Environment Variables

Environment Variable	**What It Affects**
`CDPATH, cdpath`	The alias names for directories accessed with the `cd` command
`EDITOR`	The default editor used in programs such as the e-mail program Elm
`ENV`	The path along which UNIX looks to find configuration files
`HOME, home`	The name of the user's home directory when the user first logs on
`MAIL, mail`	The name of the system mailbox file
`PATH, path`	The directories that a shell searches to find a command or program
`PS1, prompt`	The shell prompt that appears on the command line
`PWD, cwd`	The name of the current working directory
`TERM`	The type of console terminal being used

2.9 SHELL METACHARACTERS

Most of the characters other than letters and digits have special meaning to the shell. These characters are called *shell metacharacters* and, therefore, cannot be used in shell commands as literal characters without specifying them syntactically in a particular way. Thus, try not to use them in naming your files. Also, when these characters are used in commands, no space is required before or after a character. However, you can use spaces before and after a shell metacharacter for clarity. Table 2.7 contains a list of the shell metacharacters and their purposes.

TABLE 2.7 Shell Metacharacters

Metacharacter	**Purpose**	**Example**
`<New Line>`	To end a command line	
`<Space>`	To separate elements on a command line	`ls /etc`
`<Tab>`	To separate elements on a command line	`ls /etc`
`#`	To start a comment	`# This is a comment line`
`&quot;`	To quote multiple characters but allow substitution	`&quot;$file&quot; bak`
`$`	To end line and dereference a shell variable	`$PATH`
`&`	To provide background execution of a command	`command &`
`'`	To quote multiple characters	`'$100,000'`
`( )`	To execute a command list in a subshell	`(command1; command2)`
`*`	To match zero or more characters	`chap*.ps`
`[ ]`	To insert wild cards	`[a-s] or [1,5-9]`
`^`	To begin a line and negation symbol	`[^3-8]`
`'`	To substitute a command	`PS1='command'`
`{ }`	To execute a command list in the current shell	`{command1; command2}`
`|`	To create a pipe between commands	`command1 | command2`
`;`	To separate commands in sequential execution	`command1; command2`
`<`	To redirect input for a command	`command < file`
`>`	To redirect output for a command	`command > file`
`?`	To substitute a wild card for **lab.?** exactly one character	
`/`	To be used as the root directory and /usr/bin as a component separator in a pathname	
	To escape/quote a single character; `n` command `arg1\`used to quote `<New Line>` character `arg2 arg3` to allow continuation of a shell `\?` command on the following line	
C and Korn Shells Only		
`!`	To start an event specification in the history list and the current event	`!!, !$`
`%`	The C shell prompt, or the starting character for specifying a job number	`% or %3`
`~`	To name home directory	`~/.profile`

The shell metacharacters allow you to specify multiple files in multiple directories in one command line. We describe the use of these characters in subsequent chapters, but we give some simple examples here to explain the meanings of some commonly used metacharacters:

- `*`
- `?`
- `~`
- `[ ]`

The `?.txt` string can be used for all the files that have a single character before **.txt**, such as **a.txt**, **G.txt**, **@.txt**, and **7.txt**. The `[0-9].c` string can be used for all the files in a directory that have a single digit before **.c**, such as **3.c** and **8.c**. The `lab1/c` string stands for **lab1/c**. Note the use of backslash (\) to quote (escape) the slash character (/).

The following command prints the names of all the files in your current directory that have two-character file names and an .html extension, with the first character being a digit and the second being an uppercase or lowercase letter. The printer on which these files are printed is **spr**.

```
$ lpr -Pspr [0-9][a-zA-Z].html
$
```

Note that **`[0-9]`** means any digits from 0 through 9 and **`[a-zA-Z]`** means any lowercase or uppercase letter. The following command displays the names of all six-character-long files with **.c** extension in your current directory, with the first three characters being `lab`, the fourth being a digit, and the remaining being any two characters.

```
$ ls lab[0-9]??.c
lab11a.clab1a1.c lab123.clab4ab.c
$
```

2.10 THE SUDO AND SU COMMANDS

The `sudo` command allows a permitted user to execute a command as the superuser, or to assume the role of another user, as specified by security policy. The `su` command allows an ordinary user to switch user roles or to also simulate being the superuser on the system. The superuser has file permission and access privileges to everything on the system.

In many of the operations shown in the following chapters, particularly in Chapter 23 on system administration, it will be necessary to execute the `su` command in order to accomplish the tasks shown. In order to use this command, it is necessary to know the root or superuser password.

We give a more complete explanation of the sudo command in Chapter 23, Section 23.9.3.1.

An example of using the su command on a PC-BSD system is as follows:

```
[bob@pcbsd-923]% su
Password: XXX
[bob@pcbsd-923] /usr/home/bob#
```

SUMMARY

The UNIX operating system is most famous for its text-based command execution, but in the twenty-first century it has a competitively developed GUI environment as well. This chapter serves to familiarize you with the basic structure of a CUI UNIX command. It also shows you how to log in via three popular and typical login methods, and how to gracefully log off.

A beginner must be able to do basic file maintenance, and a core set of CUI file maintenance commands and their options are introduced in this chapter. These commands will be useful throughout the rest of this book. Finally, we illustrate and give examples of some basic utility commands—most importantly, the commands and their options that allow you to print files and the alias command.

When you log on to a UNIX computer, the system runs a program called a shell that gives you a prompt and waits for you to type commands, either as single commands or as multiple commands connected by redirection or piping operators. The shell program, coded in C, is an interpreter, and as such has the same structured programming capabilities of high-level languages. When you type a command and press <Enter>, the shell interprets and tries to execute the command, assuming that the first word in the command line is the name of the command. A shell command can be built-in or external. The shell has the code for executing a built-in command, but the code for an external command is in a file. To execute an external command, the shell searches several directories, one by one, to locate the file that contains the code for the command. If the file is found, it is executed if it contains code (binary or shell script). The names of the directories that the shell searches to locate the file for an external command form are known as the search path. The search path is stored in a shell variable called PATH (for the Bourne, Korn, and Bash shells) or path (for the C shell). You can change the search path for your shell by adding new directory names in PATH or by deleting some existing directory names from it.

Several shells are available for you to use. These shells differ in terms of convenience of use at the command line level and features available in their programming languages. The most commonly used shells in a UNIX-based system are the Bash and C Shells. The Bourne shell is the oldest and has a good programming language. The C shell has a more convenient and rich command-level interface. The Korn shell has some good features of both and is a superset of the Bourne shell.

Certain characters, called shell metacharacters, have special meaning to the shell. Because the shell treats them in special ways, they should not be used in file names. If you must use them in commands, you need to quote them for the shell to treat them literally.

QUESTIONS AND PROBLEMS

1. Create a directory called UNIX in your home directory. What command line did you use to do this?
2. Give a command line for displaying the files **lab1**, **lab2**, **lab3**, and **lab4**. Can you give two more command lines that do the same thing? What is the command line for displaying the files **lab1.c**, **lab2.c**, **lab3.c**, and **lab4.c**? (Hint: use shell metacharacters.)
3. Give a PC-BSD command line for printing all the files in your home directory that start with the string memo and end with **.ps** on a printer called **upmpr**. What command line did you use to do this?
4. Give the command line for nicknaming the command `who -H` as `W`. Give both Bourne and C shell versions. Where would you put it if you want it to execute every time you start a new shell?
5. Type the command man `ls > ~/UNIX/ls.man` on your system. This command will put the man page for the `ls` command in the **ls.man** file in your **~/UNIX** directory (the one you created in Problem 1). Give the command for printing two copies of this file on a printer in your lab. What command line would you use on PC-BSD to achieve this printing? What command would you use on Solaris to achieve this printing?
6. What is the `mesg` value set to for your environment? If it is on, how would you turn off your current session? How would you set it off for every login?
7. What does the command `lpr -Pqpr [0-9]*.jpg` do in PC-BSD? Explain your answer.
8. Use the `passwd` command to change your password. If you are on a network, be aware that you might have to use the `yppasswd` command to modify your network login password. Also, make sure you abide by the rules set up by your system administrator for coming up with good passwords!
9. Using the correct terminology (e.g., command, option, option argument, and command argument), identify the constituent parts of the following UNIX single commands.
   ```
   ls -la *.exe
   lpr -Pwpr file27
   chmod g+rwx *.*
   ```
10. View the man pages for each of the useful commands listed in Table 2.3. Which part of the man pages is most descriptive for you? Which of the options shown on each of the man pages is the most useful for beginners? Explain.
11. How many users are logged on to your system at this time? What command did you use to discover this?

12. Determine the name of the operating system that your computer runs. What command did you use to discover this?

13. Give the command line for displaying manual pages for the socket, read, and connect system calls on a PC-BSD system. What will be the command line for a Solaris computer?

14. What is a shell? What is its purpose?

15. What are the two types of shell commands? What are the differences between them?

16. Give names of five UNIX shells. Which are the most popular? What is a login shell? What do you type in to terminate the execution of a shell? How do you terminate the execution of your login shell?

17. What shells do you think are *supersets* of other shells? In other words, which shells have other shells’ complete command sets plus their own? Can you find any commands in a subset shell that are not in a superset shell? Refer to Figure 2.7.

18. What is the search path for a shell? What is the name of a shell variable that is used to maintain it for the Bourne, C, and Korn shells? Where (i.e., in which file) is this variable typically located?

19. What is the search path set to in your environment? How did you find out? Set your search path so that your shell searches your current and your ~/**bin** directories while looking for a command that you type. In what order does your shell search the directories in your search path? Why?

20. What are hidden files? What are the names of the hidden files that are executed when you log on to System V and BSD UNIX systems?

21. What is a shell start-up file? What is the name of this file for the C shell? Where (i.e., in which directory) is this file stored?

22. What important features of each shell, as discussed on the manual pages for that shell, seem to be most important for you as a new, intermediate, or advanced user of UNIX? Explain the importance of these features to you in comparison with the other shells available and their features.

23. Suppose that your login shell is a C shell. You receive a shell script that runs with the Bourne shell. How would you execute it? Clearly write down all the steps that you would use.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

CHAPTER 3

Editing Text Files

Objectives

- To explain the general utility of editing text files on a UNIX system
- To show the basic capabilities of vi, vim, and gvim, and how to customize them
- To show the basic capabilities of GNU emacs and how to customize it
- To cover the commands and primitives

  `cp, emacs, gvim, ls, pwd, sh, vi, vim, who`

3.1 INTRODUCTION AND QUICK START

In this chapter, we use the following editors that are commonly available in both of our base modern UNIX systems, PC-BSD and Solaris: vi, vim, gvim, and GNU emacs.

3.1.1 Quick Start: The Simplest Path through These Editors

To stress how the keyboard keys are used in these editors, we provide the following reference to the keys used to execute commands or change modes:

1. Pressing the Escape key is signified as `<Esc>`
2. Pressing the Enter key is signified as `<Enter>`
3. Pressing the `<Ctrl>` key in combination with another single key is signified as `<Ctrl+X>`, where you hold down the `<Ctrl>` key and press the `X` key (or any valid key for that combination) <u>at the same time</u>.
4. Pressing the Alt key in combination with another single key is signified as `<Alt+X>`, where you hold down the `<Alt>` key and press the `x` key (or any valid key for that combination) <u>at the same time</u>.

5. A variant of 3. and 4. is shown as `<Ctrl+X> a [b]`, where you first press and release `<Ctrl>` and `x` simultaneously, then press the `a` key, and optionally press the `b` key (or any valid combination of single keys or strings of characters).
6. In GNU emacs for PC-BSD and Solaris, the `Meta` key that is referred to in much of the literature on GNU emacs is the `<Alt>` key.

What you type or hold down on the keyboard is shown in **`bold`** text.

3.1.1.1 For vi, vim, and gvim

- At the shell prompt, run the program by typing **`vi file1`** then press **`<Enter>`**.
- Type **`A`**.
- Type some text.
- Press **`<Esc>`**.
- Type **:** (colon).
- Type **`wq`** then press **`<Enter>`**.

You now have a file in your default directory named **file1** with the text you typed in it.

If GNU emacs is not installed on your system, skip ahead to the next subsection for general instructions on how to install it. Then do the following:

3.1.1.2 For GNU emacs

- At the shell prompt, run the program by typing **`emacs file2`** then press **`<Enter>`**.
- Type some text.
- Hold down **`<Ctrl+U>`**, then **`<Ctrl+X>`**, then **`<Ctrl+C>`**.

You now have a file in your default directory named **file2** with the text you typed in it.

3.1.2 First Comments on UNIX Editors

As you can see from Section 3.1.1, with vi, vim, and gvim, you can't immediately begin to enter text into the file you are editing. You have to be in *Insert mode* to do that; that's what typing `A` as the second step is doing. Vi, vim, and gvim have modes.

In GNU emacs, you can start typing text into the file immediately. Emacs is a *modeless* editor.

We present the tutorial information in this chapter using typed commands, and by using graphical modes of input and editing.

It is very important to realize that vi, vim, and gvim all generally use the same commands and have basically the same functionality. But vim and gvim are not only more graphical—allowing you to work more efficiently in GUI environments such as those on our base modern UNIX systems, PC-BSD and Solaris—but they also have an improved and expanded command structure. This will become more evident to you, for example, in Section 3.2.9, where vim has special improved macro-writing capabilities that vi does not.

At the time of writing, both PC-BSD and Solaris have vi, vim, and gvim preinstalled if you have done a basic installation of the system as detailed at the beginning of Chapter 23. But GNU emacs is not preinstalled. Therefore, you must look ahead to Chapter 23, Section 23.7, "System Updates and Software Upgrades by Using a Package Manager." In that section, you will be shown how to use the appropriate package management facilities to obtain GNU emacs for your system. In the App Café in PC-BSD, you must browse **Categories>Editors for Emacs** and install it. From the Solaris IPS repository, you must browse **Development>Editors**, and install `gnu-emacs`.

We will not cover the details of the installation of any of the editors we demonstrate here. In addition, if you are logging into a UNIX system via a terminal window, such as with PuTTY from a Windows machine, many of the graphical modes and techniques shown in this chapter will not be available to you. But that does not prevent you from using the traditional typed commands and keyboard edits that we show.

3.1.3 Using Text Editors

Modern UNIX uses both a GUI, with powerful window management systems like Gnome and KDE, and a CUI. Therefore, to do useful things such as execute multiple commands from within a script file, write e-mail messages, or create C language programs, you must be familiar with one or perhaps multiple ways of entering text into a file. In addition, you must also be familiar with how to edit existing files efficiently—that is, to change their contents or otherwise modify them in some way. Text editors allow you to view a file's contents, similar to the `more` command, so that you can identify the key features of the file, and then read and utilize the information contained in it. For example, a file without any extension, such as **foo** (rather than **foo.txt**) might be a text file that you can view with a text editor.

The editors that we consider here are all considered full-screen display editors. That is, on the display screen or monitor that you are using to view or edit a file, you are able to see a portion of the file, which fills most or all of the window allocated to the text editor screen display. You are also able to move the cursor, or point, to any of the text you see in this full-screen display, with either the arrow keys on the keyboard or with a mouse. That text material is usually held in a temporary storage area in computer memory called the editor *buffer*. If your file is larger than one screen, the buffer contents change as you move the cursor through the file. The difference between a file, which you edit, and a buffer is crucial. For text-editing purposes, a file is stored on disk as a sequence of data. When you edit that file, you edit a copy that the editor creates, which is in the editor buffer. You make changes to the contents of the buffer—and can even manipulate several buffers at once—but when you save the buffer, you write a new sequence of data to the disk, thereby saving the file.

TABLE 3.1 Basic Text-Editing Functions

Function	Description
Cursor movement	Moving the location of the insertion point or current position in the buffer
Cut or copy, paste	"Ripping out" text blocks or duplicating text blocks, reinserting ripped or duplicated blocks
Deleting text	Deleting text at a specified location or in a specified range
Inserting text	Placing text at a specified location
Opening, starting	Opening an existing file for modification, beginning a new file
Quitting	Leaving the text editor, with or without saving the work done
Saving	Retaining the buffer as a disk file
Search, replace	Finding instances of text strings, replacing them with new strings

Another important operational feature of all the editors discussed in this chapter is that, traditionally, their actions are based on keystroke commands, whether they are a single keystroke or combinations of keys pressed simultaneously or sequentially. Because one of the primary input devices in UNIX is the keyboard, using the correct syntax of keystroke commands is mandatory. But the keyboard method of input, once you have become accustomed to it, is as efficient or, for some users, even more efficient than mouse/GUI input. Keystrokes also are more flexible, giving you more complete and customizable control over editing actions. Generally, you should choose the editor you are most comfortable with, in terms of the way you prefer to work with the computer. However, your choice of editor also depends on the complexity and quantity of text creation and manipulation that you want to do. Practically speaking, editors such as vi, vim, gvim, and GNU emacs are capable of handling complex editing tasks in multiple windows on multiple files, and provide you with a visual software development environment, as well as document production and management capability. But to take advantage of that power, you have to learn the mechanics of the commands that are needed to perform those tasks and how they are implemented either graphically or by typing them—and retain that knowledge. The basic functions common to the text editors that we cover here are listed in Table 3.1, along with a short description of each function.

3.2 USING THE vi, vim, AND gvim EDITORS

The vi, vim, and gvim UNIX text editors have almost all the features of a word processor and have tremendous flexibility in creating text files. They are complex to learn, but their advantages give you the ability to create, manipulate, and use the kinds of text files that the full range of UNIX users, from absolute novice to seasoned veteran, commonly work with. We will proceed in the following section and subsections by demonstrating vi as a text-only interface editor, then move to a more graphical interface approach with vim and gvim.

Buffers: As we mentioned in Section 3.1, the notion of a *buffer* as a temporary storage facility for the text that you are editing is very useful and important in vi, vim, and gvim. The main buffer, sometimes referred to as the editing buffer or the work buffer, is the main repository for the body of text that you are trying to create or to modify from some previous permanently archived file on disk. The general purpose buffer is where your most recent "ripped-out" (cut/copied) text is retained. Indexed buffers allow you to store more than one temporary string of text.

3.2.1 Basic Shell Script File Creation, Editing, Execution

Shell Script File: Practice Session 3.1 shows how to create a script file, or collection of UNIX commands that are executed in sequence, and then execute the script. We present more about shell programming and script files in Chapters 12 through 15. For this example, we assume that you are running the Bourne shell. If you are running some other shell by default, go back to Chapter 2, Section 2.8, and review how to identify and change shells.

In PC-BSD, to run an interactive shell, such as the Bourne shell shown in Practice Session 3.1, on top of your login shell (which is the C shell by default in that system), at the C shell prompt you type `sh` and press `<Enter>`.

If you are using Solaris, do Practice Session 3.1 for the Bourne Again (Bash) shell, which is the default shell in that system, and don't change shells.

And do not worry too much if you make an error in Steps 2, 3, and 4; you can go through the rest of the script file discussion and then come back to this example after you have learned some of the editing commands and become more familiar with them.

3.2.1.1 Practice Session 3.1

Step 1: At the shell prompt, start vi by typing **`vi firscrip`** and then pressing <**Enter**>. The vi screen appears on your display.

Step 2: Type A. Then type **`ls -la`** and then press **`<Enter>`**.

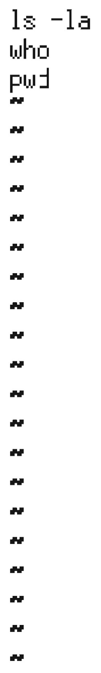
Step 3: Type **`who`** and then press **`<Enter>`**.

Step 4: Type **`pwd`** and then press the **`<Esc>`** key. At this point, your screen should look like that shown in Figure 3.1.

Step 5: Type **`:wq`** and then press **`<Enter>`**.

Step 6: At the shell prompt, type **`sh firscrip`** and then press **`<Enter>`**.

Step 7: Note the results. How many files do you have in your present working directory? What are their names and sizes? Who else is using your computer system? What is your present working directory?



```
ls -la
who
pwd
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
```

FIGURE 3.1 File **firscrip** after Step 4.

In Practice Session 3.1, you accomplished these things:

- At Step 2, typing `A` took vi out of *Command mode* (which is what vi starts in by default) and placed it in one of the forms of Insert mode. In other words, anything that you typed at the keyboard was appended as text on the first line in the text area of the editor.
- When you pressed the `<Esc>` key in Step 4, vi was taken out of Insert mode and put back into Command mode.
- When you typed `:` in Step 5, that was a valid Command mode prefix character for the two commands that followed, and put vi in *Last Line mode*.
- When you typed `wq` after the `:`, vi interpreted those commands in Last Line mode as write out or save the file, and quit the editor.
- Step 6 executes the Bourne shell script file.

3.2.2 How to Start, Save a File, and Exit

When you need to do UNIX text editing that gives you as much functionality as a typical word processor, you can use the vi text editor. To start vi from the command line, use the following general syntax (anything enclosed in square brackets `[ ]` is optional):

SYNTAX

`vi [options] [file(s)]`

Purpose: Allows you to edit a new or existing text file(s)

Output: With no options or file(s) specified, you are placed in the vi program and can begin to edit a new buffer

Commonly used options/features:

`+n`	Begin to edit file(s) starting at line number `n`
`+/exp`	Begin to edit at the first line in the file matching string `exp`

The operations that you perform in vi fall into two general categories: *Command mode* operations, which consist of key sequences that are commands to the editor to take certain actions, and *Insert mode* operations, which allow you to input text.

The general organization of the vi text editor and how to start, exit, and switch modes are illustrated in Figure 3.2. The general organization of vim and gvim, and how to start, exit, and switch modes in those editors, is the same as shown for vi in Figure 3.2.

For example, to change from Command mode, which you are in when you first enter the editor, to Insert mode, type a valid command, such as `A` to append text at the end of the current line. Certain commands that are prefixed with the `:`, `/`, `?`, or `:!` characters are echoed or shown to you on the last line on the screen and must be terminated by pressing `<Enter>`. *Last Line mode*, sometimes called *ex mode* because it is derived from the ex editor, allows you to execute certain commands and leave the editor. To change from Insert mode to Command mode, press the `<Esc>` key.

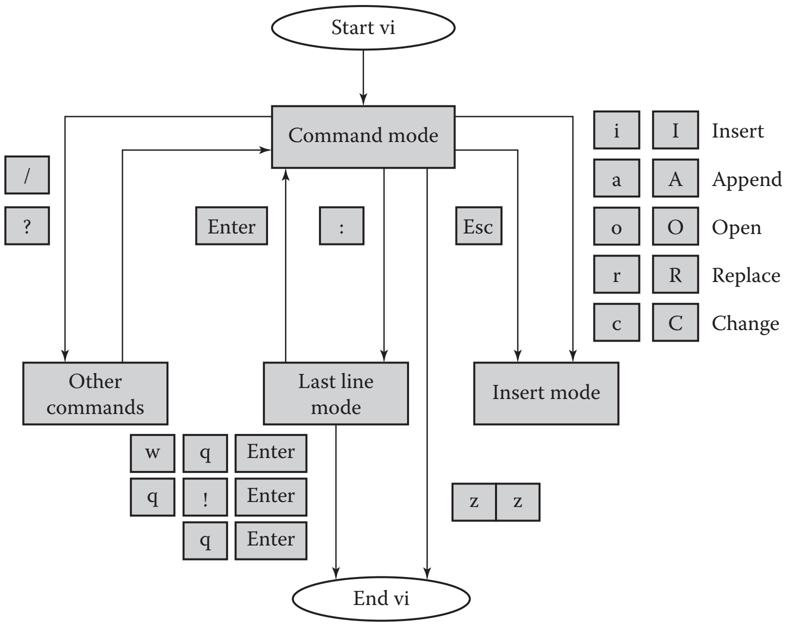


FIGURE 3.2 General organization of vi, vim, and gvim.

The keystroke commands that you execute in vi are case sensitive; for example, uppercase `A` appends new text after the last character at the end of the current line, whereas lowercase `a` appends new text after the character the cursor is on.

To start vi, at the shell prompt, type `vi` (and optionally designate some option[s] and file name[s]) and then press `<Enter>`. You are now in Command mode. To enter Insert mode, type `A` and you are now able to insert text on the first line of the file.

After entering text, you can press the `<Esc>` key to enter Command mode.

At any point in your creation or manipulation of text, you can press the `u` key on the keyboard to undo the last operation.

From Command mode, you can save the text that you just inserted into the buffer to a file on disk by typing `:w filename` and pressing `<Enter>`, where `filename` is the name of the file you want to save the text to. To quit the editor, type `:q`.

3.2.3 The Format of a vi Command and the Modes of Operation

In Command mode, the generic syntax of keystrokes is:

```
[#1] operation [#2] target
```

where:

anything enclosed in [ ] is optional;

`#1` is an optional number, such as 5, specifying how many operations are to be done;

`operation` is what you want to accomplish, such as deleting lines of text;

`#2` is an optional number, such as 5, specifying how many targets are affected by the **`operation`**; and

`target` is the text that you want to do the operation on, such as an entire line of text.

Note that if the current line is the target of the operation, the syntax for specifying the target is the same as the syntax of the operation; for example, `dd` deletes the current line. Also, a variation on this generic syntax is the cursor movement command, whereby you can omit the numbers and operation and simply move the cursor by word, sentence, paragraph, or section. Table 3.2 lists some specific examples of this generic syntax and variations used in Command mode.

As previously stated, when you start vi, it is in Command mode. When you want to be in Insert mode instead of Command mode, press a valid key to accomplish the change. Some of these keys are shown in Table 3.3.

After inserting text, you can edit the text, move the cursor to a new position in the buffer, and save the buffer and exit the editor—all from within Command mode. When you want to change from Insert mode to Command mode, press the `<Esc>` key.

To save the buffer and exit the editor, press the `:` key (colon) to enter Last Line mode. The general commands that are useful in Last Line mode are shown in Table 3.4.

TABLE 3.2 Examples of Vi Command Syntax

Command	Action
`cw`	Change word.
`cc`	Change line.
`c$`	Change text from current position to end of line.
`C`	Same as `c$`.
`dd`	Delete current line.
`7 dd`	Delete 7 lines.
`d$`	Delete text from current position to end of line.
`D`	Same as `d$`.
`5dw`	Delete 5 words.
`d7,14`	Delete lines 7 through 14 in the buffer.
`d}`	Delete up to next paragraph.
`d^`	Delete back to beginning of line.
`d/ pat`	Delete up to first occurrence of pattern.
`dn`	Delete up to next occurrence of pattern.
`df x`	Delete up to and including x on current line.
`dt x`	Delete up to (but not including) x on current line.
`dL`	Delete up to last line on screen.
`dG`	Delete to end of file.
`gqap`	Reformat current paragraph to text width (vim and gvim).
`g~w`	Switch case of word (vim and gvim).
`guw`	Change word to lowercase (vim and gvim).
`gUw`	Change word to uppercase (vim and gvim).
`p`	Insert last deleted or yanked text after cursor.
`gp`	Same as `p`, but leave cursor at end of inserted text (vim and gvim).
`gP`	Same as `P`, but leave cursor at end of inserted text (vim and gvim).
`]p`	Same as `p`, but match current indention (vim and gvim).
`[p`	Same as `P`, but match current indention (vim and gvim).
`P`	Insert last deleted or yanked text before cursor.
`r x`	Replace character with x. Does not require the use of `<Esc>`!
`R text`	Replace with new text (overwrite), beginning at cursor. `<Esc>` ends replace mode.
`s`	Substitute character. `<Esc>` ends substitute mode.
`4s`	Substitute four characters. `<Esc>` ends substitute mode.
`S`	Substitute entire line. `<Esc>` ends substitute mode.
`u`	Undo last change.
`<Ctrl+R>`	Redo last change (vim and gvim).
`U`	Restore the current line, if you have not moved off of it.
`x`	Delete current cursor position.
`X`	Delete back one character.
`5X`	Delete previous 5 characters
`.`	Repeat last change.
	Change case and move cursor right.
`<Ctrl+A>`	Increment number at the cursor (vim and gvim).
`<Ctrl+X>`	Decrement number at the cursor (vim and gvim).

TABLE 3.3 Important Keys to Switch from Command Mode to Insert Mode

Key	Action
`a`	Appends text after the character the cursor is on
`A`	Appends text after the last character of the current line
`c`	Begins a change operation, allowing you to modify text
`C`	Changes from the cursor position to the end of the current line
`i`	Inserts text before the character the cursor is on
`I`	Inserts text at the beginning of the current line
`o`	Opens a blank line below the current line and puts the cursor on that line
`O`	Opens a blank line above the current line and puts the cursor on that line
`R`	Begins overwriting text
`s`	Substitutes single characters
`S`	Substitutes whole lines

TABLE 3.4 Important Commands for Command Mode

Command	Action
`: n, m w file`	Write lines `n` to `m` to new file.
`: n, m w >> file`	Append lines `n` to `m` to existing file.
`:r filename`	Reads and inserts the contents of the file filename at the current cursor position
`:wq`	Saves the buffer and quits
`:w`	Saves the current buffer and remains in the editor.
`:w filename`	Saves the current buffer to filename
`:w! filename`	Overwrites filename with the current text
`:w!`	Write file (overriding protection).
`:w! file`	Overwrite file with current text.
`:w %.new`	Write current buffer named **file** as **file.new.**
`:q`	Quit vi (fails if changes were made).
`:q!`	Quit vi without saving the buffer.
`:Q`	Quit vi and invoke `ex`.
`:vi`	Return to vi after `Q` command.
`ZZ`	Quits vi, saving the file only if changes were made since the last save
`%`	Replaced with current filename in editing commands.
`#`	Replaced with alternate filename in editing commands.

For now, we recommend that you use the arrow keys on the keyboard to move the cursor around in the buffer. It is possible to also use the `h`, `j`, `k`, and `l` keys on the keyboard to move the cursor. In gvim, you can use the mouse and its buttons!

The following practice session introduces you to some of the commands presented in Tables 3.2 through 3.4:

3.2.3.1 Practice Session 3.2

Step 1: At the shell prompt, type **`vi firstvi`** and then press `<Enter>`.

Step 2: Type `A`, then type **`This is the first line of a vi file.`** and then press **`<Enter>`**.

Step 3: Type **`This is the line of a vi file.`** and then press **`<Enter>`**.

Step 4: Type **`is the 3r line of a vi.`**

Step 5: Press the **`<Esc>`** key.

Step 6: Type **`:w`** and then press **`<Enter>`**.

Step 7: Use the arrow keys on the keyboard to position the cursor on the character `l` in the word `line` on the second line of the file.

Step 8: Type **`i`** and then **`2nd`**.

Step 9: Press the **`<Esc>`** key.

Step 10: Use the arrow keys to position the cursor anywhere on the third line of the file.

Step 11: Type I and then **`This`**.

Step 12: Press the **`<Esc>`** key.

Step 13: Use the arrow keys on the keyboard to position the cursor on the character `r` in `3r` on this line.

Step 14: Type **`a`** and then **`d`**.

Step 15: Press the **`<Esc>`** key.

Step 16: Type A and then **`file`**.

Step 17: Press the **`<Esc>`** key on the keyboard. Your screen display should look similar to Figure 3.3.

Step 18: Type **`:wq`**. You will be back at the shell prompt.

The following in-chapter exercise asks you to apply some of the operations you learned about in the previous practice session.

EXERCISE 3.1

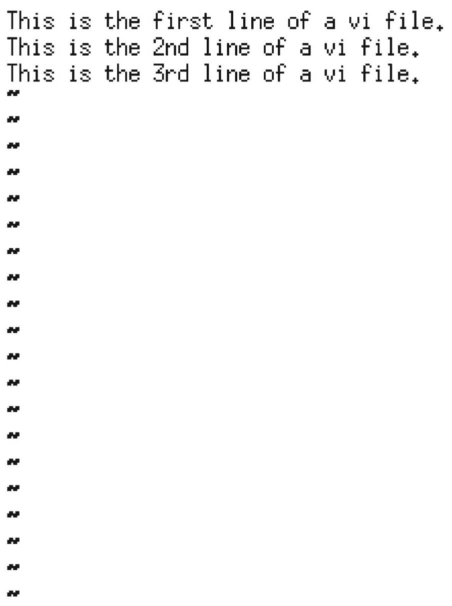
With vi you begin editing a file that you created yesterday. You want to save a copy of it with a different filename while still in vi, but you don't want to quit this editing session. How do you accomplish this result in vi?

EXERCISE 3.2

What happens if you accomplish five operations in vi and then type `5u` when in Command mode?

3.2.4 Cursor Movement and Editing Commands

In Command mode, several commands accomplish cursor movement and text-editing tasks. Table 3.5 lists important cursor movement and keyboard editing commands. As



```
This is the first line of a vi file.
This is the 2nd line of a vi file.
This is the 3rd line of a vi file.
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
```

FIGURE 3.3 File **firstvi**.

TABLE 3.5 Cursor Movement and Keyboard Editing Commands

Command	Action
`1G`	Moves the cursor to the first line of the file
`G`	Moves the cursor to the last line of the file
`0` (zero)	Moves the cursor to the first character of the current line
`<Ctrl+G>`	Reports the position of the cursor in terms of line # and column #
`$`	Moves the cursor to the last character of the current line
`w`	Moves the cursor forward one word at a time
`b`	Moves the cursor backward one word at a time
`x`	Deletes the character at the cursor position
`dd`	Deletes the line at the current cursor position
`u`	Undoes the most recent change
`r`	Replaces the character at the current cursor location with what is typed next

we have already noted, character-at-a-time or line-at-a-time moves of the cursor can be accomplished easily with the arrow keys, or alternatively with the `h`, `j`, `k`, and `l` keys on the keyboard.

The following practice session lets you continue editing the file you created in Practice Session 3.2 by using commands presented in Table 3.5.

3.2.4.1 Practice Session 3.3

Step 1: At the shell prompt, type **`vi firstvi`** and then press **`<Enter>`**.

Step 2: Type **`G`**. The cursor moves to the last line of the file.

Step 3: Hold down the **`<Ctrl>`** and **`g`** keys at the same time. On the last line of the screen display, vi reports the following:

```
"firstvi" line 3 of 3 - - 100%-- col 1
```

This is a report of the buffer that you are editing, the current line number, the total number of lines in the buffer, the percentage of the buffer that this line represents, and the current column position of the cursor.

Step 4: Type **`o`**. A new line opens below the third line of the file

Step 5: Type **`This is the 5th line of a vi file.`** Type **`<Esc>`**.

Step 6: Type **`0`** (zero). The cursor moves to the first character of the line you just typed in.

Step 7: Type **`$`**. The cursor moves to the last character of the current line.

Step 8: Type **`O`**. A new line opens above the current fourth line.

Step 9: Type **`This is the 44th line of a va file.`** Type **`<Esc>`**.

Step 10: Use the arrow keys to position the cursor over the first `4` in `44` on this line.

Step 11: Type **`x`**.

Step 12: Use the arrow keys to position the cursor over the `a` in `va` on this line.

Step 13: Type **`r`** and then type **`i`**.

Step 14: Type **`dd`**.

Step 15: Type **`:wq`** to go back to the shell prompt.

Step 16: At the shell prompt, type **`more firstvi`** and then press **`<Enter>`**. How many lines with text on them does **`more`** show in this file?

3.2.5 Yank and Put (Copy and Paste) and Substitute (Search and Replace)

Every word processor is capable of copying and pasting text and also of searching for old text and replacing it with new text. Copying and pasting are accomplished with the vi commands `yank` and `put`. In general, you use yank and put in sequence and move the cursor (with any of the cursor movement commands or methods) only between yanking and putting. Some examples of the syntax for yank and put are given in Table 3.6.

The simple vi forms of search and replace are accomplished using the `substitute` command. This command is executed when vi is in Last Line mode, where you preface the

TABLE 3.6 Examples of Syntax for the Yank and Put Commands

Command Syntax	What It Accomplishes
`y2W`	Yanks two words, starting at the current cursor position, going to the right
`4yb`	Yanks four words, starting at the current cursor position, going to the left
`yy or Y`	Yanks the current line
`p`	Puts the yanked text after the current cursor position
`P`	Puts the yanked text before the current cursor position
`5p`	Puts the yanked text in the buffer five times after the current cursor position
`Y`	Copy current line
`yy`	Copy current line
`" x yy`	Copy current line to register `x`
`ye`	Copy text to end of word
`yw`	Like `ye`, but include the whitespace after the word
`y$`	Copy rest of line
`&quot; x dd`	Delete current line into register `x`
`&quot; x d`	Delete into register `x`
`&quot; x p`	Put contents of register `x`
`y]]`	Copy up to next section heading
`J`	Join current line to next line
`gJ`	Same as `J`, but without inserting a space (vim and gvim)
`:j`	Same as `J`
`:j!`	Same as `gJ`

command with the `:` character and terminate the command by pressing `<Enter>`. The format of the `substitute` command as it is typed on the status line is:

```
:[range]s/pattern/string[/option(s)][count]
```

where:

anything enclosed in **`[ ]`** is not mandatory;

`:` is the colon prefix for the Last Line mode command;

`range` is a valid specification of lines in the buffer (or the current line is the **`range`**);

`s` or **`substitute`** is the syntax of the substitute command;

/ is a delimiter for searching;

`pattern` is the text or objects you want to replace;

/ is a delimiter for replacement;

`string` is the new text or objects;

TABLE 3.7 Examples of Syntax for the Substitute Command

Command Syntax	What It Accomplishes
`:s/john/jane/`	Substitutes the word `jane` for the word `john` on the current line, only once.
`:s/john/jane/g`	Substitutes the word `jane` for every word `john` on the current line.
`:1,10s/big/small/g`	Substitutes the word `small` for every word `big` on lines 1–10.
`:1,$s/men/women/g`	Substitutes the word `women` for every word `men` in the entire file.
`:'<,'>s/this/that/g`	Select the range in Command mode first by typing `<Ctrl+V>` and using the arrow keys. Then type **:**. The word `that` will be substituted for the word `this` (vim, gvim only).
`:s/ \<tim\>/tom/`	Substitutes only the whole word `tim` with the word `tom`, not the partial match of `tim` in any string.
`:%s/terrible/wonderful/gc`	Interactive substitution using c option of the word `terrible` with the word `wonderful` (vim, gvim only).
`:%s/^/ \=line(".") . ". "/g`	Makes the line numbers of all lines in the buffer permanently part of each line (vim, gvim only).

`/option(s)` is a modifier, usually **`g`** for global, to the command; and

`count` is the number of lines to execute the command on from the current position.

The grammar of `pattern` and `string` can be extremely explicit and complex, and may take the form of a regular expression. (We present more information on the formation of regular expressions in Chapter 7, Section 7.2.) Some examples of the syntax for the `substitute` command, including vim/gvim-only constructions, are given in Table 3.7.

Practice Session 3.4 shows you how to use the vi commands `yank` and `put` to copy and paste. It also allows you to do individual and multiple searches and replace text with the vi `substitute` command.

3.2.5.1 Practice Session 3.4

Step 1: At the shell prompt, type **`vi multiline`** and then press **`<Enter>`**.

Step 2: Type **`A`** and then type **`Windows is the operating system of choice for everyone.`**

Step 3: Press the **`<Esc>`** key. You have left Insert mode and are now in Command mode.

Step 4: Press the **`0`** (zero) key. The cursor moves to the first character of the first line.

Step 5: Type **`yy`**. This action yanks, or copies, the first line to a special buffer.

Step 6: Type **`7p`**. This action puts, or pastes, the first line seven times, creating seven new lines of text containing the same text as the first line. The cursor should now be on the first character of the eighth line.

Step 7: Type **1G**. This action puts the cursor on the first character of the first line in the buffer.

Step 8: Hold down the **<Shift>** and **;** keys at the same time. Doing so places a **:** in the status line at the bottom of the vi screen display, allowing you to type a command.

Step 9: Type **s/everyone/students/** and then press **<Enter>**. The word `everyone` at the end of the first line is replaced with the word `students`.

Step 10: Use the arrow keys to position the cursor on the first character of the second line.

Step 11: Type **:s/everyone/computer scientists/** and then press **<Enter>**.

Step 12: Repeat Steps 8–10 on the third through eighth lines of the buffer, substituting the words `engineers`, `system administrators`, `web servers`, `scientists`, `networking`, and `mathematicians` for the word `everyone` on each of those six lines.

Step 13: Type **:1,$s/Windows/UNIX/g** and then press **<Enter>**. You have globally replaced the word `Windows` on all eight lines of the file with the word `UNIX`. Correct?

Step 14: Type **:wq**. You have now saved the changes and exited from vi.

3.2.6 vim and gvim

Vim and gvim are two examples among many of enhanced, "improved" versions of vi. The following subsections illustrate some of the advantages of using vim and gvim over the traditional vi editor.

3.2.6.1 Vim Enhancements

The following capabilities of vim that enhance vi functionality, particularly the first one shown, are suggestions that you can use to expedite your editing tasks with vim and gvim over and above the capabilities of vi:

* vimrc

If you want to enable any of the improved facilities of vim and gvim, you should create a **~/.vimrc** file. Even if this file is empty, it will enable the facilities that we illustrate in this section!

* Help

In vim Last Line mode, type `help` or press the `F1>` function key.

Vim opens a help buffer that gives you extensive help on its facilities. In Last Line mode, when in the help buffer, type `q` to exit help.

* Multiple Windows

The Last Line mode command `split` splits the current window in two. You can then move the cursor up to a window with `<Ctrl+W> j` and down a window with `<Ctrl+W> k`. For example, the Last Line mode command `split new.c` splits the window and begins editing the file named **new.c**. To close a window, use the normal vim exit commands `ZZ` or `:q!`.

* Multiple Levels of Undo

Unlike vi, you can use the undo command to undo several steps back in the command history. For example, typing `u` in Command mode undoes the last action in vim, and typing `3u` in Command mode undoes the last three actions you did in vim. The undo level is set by default to 1000. You can redo multiply as well, using `<Ctrl+R>`. For example, `3 <Ctrl+R>` redoes the last 3 actions that were undone with `u`.

* Visual Mode

Typing `v` causes vim to enter *Visual mode*. You can then highlight a block of text and execute a vim Command mode operation on it. The `v` command selects text by character. The `<Ctrl+V>` command selects text as a block. The `V` command selects the current line. See Section 3.2.6.2 for more details on this facility in vim.

* The `incsearch` and `hlsearch` Environmental Options (Incremental Search and Highlight Search)

For the incremental search, by default, searching starts after you enter the string. With the option:

```
:set incsearch
```

incremental searches will be done. The vim editor will start searching when you type the first character of the search string. As you type in more characters, the search is refined.

For the highlight search option, setting the option turns on search highlighting. This option is enabled by the command:

```
:set hlsearch
```

After the option is enabled, any search highlights the string matched by the search.

* The `cindent` Environmental Option and the = Command Option

Like vi autoindent, the vim editor does a more specific form of indentation. The `cindent` option is set with the command:

```
:set cindent
```

This turns on C style indentation. Each new line will be automatically indented the correct amount according to the C indentation standard.

* The `:make` Command

To compile a C program with an accompanying make file, and correct the errors, you can type this command in Last Line mode:

```
make
```

This runs the `make` command and captures the output. When the command finishes the editor starts editing the first file. The next step is to fix the error. After that you need to go to the line causing the next error. This is done using the command:

```
cn
```

This command will go to the location of the next error even if it is in another file.

You can continue fixing problems and using `cn` until all your problems are over or you want to do a recompile. If you want to see the current error message again, use the command:

```
cc
```

* Last Line Mode Command History

When you are in Last Line mode, you can use the `<Up>` arrow key to recall an older command line entry, and then can use the `<Down>` arrow key to go forward to newer commands. Then, when you press `<Enter>` after you have indexed to that previous command in the history, that previous command is executed again.

There are four histories you can utilize in vim, but the two most important ones are for:

- Last Line mode command history
- `/` and `?` search command history

Your search history is most useful to you, particularly because if you type complex search criteria, you do not want to have to retype them every time you want to repeat that search!

The two other histories are for expressions and input lines for the `input()` function.

As an example, you have done a Last Line mode command, typed five more Last Line mode commands, and then want to repeat the first command again. To do this, in Last Line mode press the `<Up>` arrow key five times. Another way of doing this is to type the first few letters of the Last Line mode command you want to return to.

The `<Up>` key will use the text typed so far and compare it with the lines in the history. Only matching lines will be used.

If you do not find the line you were looking for, use the `<Down>` arrow key to go back to what you typed and correct that. You can also type `<Ctrl+U>` to start all over again.

To see all the lines in your Last Line mode command history, while in Last Line mode, type:

`history`

You will then see a complete history of the Last Line mode commands for this session at the bottom of the screen display.

Your entire search history for this session is displayed by typing `history/` in Last Line mode.

`<Ctrl+P>` will work like the `<Up>` arrow key, except that it doesn't matter what you already typed. `<Ctrl+N>` works like the `<Down>` arrow key.

* The Last Line Mode Command Line Window

Typing any text in the Last Line mode command history to modify a previous command and then execute it is possible but difficult.

A better way to use a modified form of a Last Line mode command from the history is to open the *command line window* while in Command mode by typing:

`q:`

Vim now opens a small utility window at the bottom of the screen. It contains the command line history and an empty line at the end, similar to this illustration:

```
+-------------------------------------+
|other window |
|~ |
|first.c==================================|
|:w second.c |
|:w third.c |
|:w fourth.c |
|:w fifth.c |
|:w sixth.c |
|:history
|: |
|command–line============================|
||
~/project.c
+-------------------------------------+
```

In the buffer in this small utility window, you are in Insert mode, and can use Insert mode commands to modify text and also move commands. You can use the arrow keys to move around.

For example, move up the history tree with the `<Up>` arrow key to the `:w third.c` line.

Change the word `third` to `thirteenth`.

Now press `<Enter>` when on that line, and this command will be executed. The command line window will then close. The `<Enter>` command will execute the line under the cursor. This works if vim is in Insert mode or in Command mode.

Unfortunately, changes you make in the command line window are lost! They do not result in any changes in the command history itself, but the command you execute when you are in the command line window will be added at the end of the history, similar to all other executed commands. Also, only one command line window can be open at a time.

The command line window is very useful when you want to see your old command history, index to a particular command, edit it, and execute it.

A search command in your history can be used to find something new if you index to it and modify it. For example, if in the command line window one of the lines contained `:s/everyone/computer scientists/`, you could index to it in the command line window and modify and execute it.

* Word Completion

When you are typing and you enter a partial word, you can cause vim to search for a completion by using `<Ctrl+P>` (search for previous matching word) and `<Ctrl+N>` (search for next match).

* Record and Playback

The `.` (period) command repeats the previous change in Command mode. To accomplish multiple, complex changes in vim Command mode, you can use the *record and playback* facility. There are three steps in record and playback:

1. The `q(register)` command starts recording keystrokes into the key named `register`. The register name must be a letter of the alphabet.
2. Type the commands you want to record in the register.
3. To end recording, press `q`.

You can now execute the macro by typing the command `@register`. For example, you have a list of filenames in a buffer that looks like this:

```
stdio.h

fcntl.h

unistd.h

stdlib.h
```

And what you want is the following:

```
#include "stdio.h"

#include "fcntl.h"
```

```
#include "unistd.h"

#include "stdlib.h"
```

You start by moving to the first character of the first line. Next, in Command mode, you execute the following commands:

```
qa
^
i#include "<Esc>
$
a"<Esc>
j
q
```

These commands do the following:

1. Start recording a macro in register `a`.
2. Move to the beginning of the line.
3. Insert the string `#include "` at the beginning of the line.
4. Move to the end of the line.
5. Append the double quotation mark (`"`) character to the end of the line.
6. Go to the next line.
7. Stop recording the macro.

Now that you have done the work once, you can repeat the change by typing the command `"@a"` three times.

The `"@a"` command can be preceded by a count, which will cause the macro to be executed that number of times. In this case you would type: `"3@a"`.

EXERCISE 3.3

How would you open a unique history window for the `/` and `?` commands?

EXERCISE 3.4

Where does the cursor have to be positioned in the buffer if you want to execute a modified version of the substitute command `:s/everyone/computer scientists/` correctly?

EXERCISE 3.5

Can you include Last Line mode commands, such as `substitute`, or write to a file, in a record and playback session?

3.2.6.2 Vim Visual Mode

Because vi does not have a graphical, or "visual," method of selecting and operating on blocks of text, we use vim Visual mode. In vim, Visual mode is the graphical and easy way to select a block of text in order to use a prescribed operator on it. The following will briefly describe Visual mode's features and give a simple example. Vim Visual mode allows you to apply commands to blocks of text that can be selected graphically, even though you may not be in a GUI environment. In general, all of the vi commands and operating modes shown previously work in both vim and gvim.

Using Visual mode is done in three steps:

Step 1: Move the cursor to the start of the text block, mark the start of the block with `"v"` (character mode), `"V"` (line mode), or `<Ctrl+V>` (blockwise mode). The character under the cursor will be used as the start of the block.

Step 2: Depending on what kind of functionality is provided in the terminal or console window you are working in, move to the end of the text block, either with the arrow keys on the keyboard, with the `h`, `j`, `k`, or `l` keys on the keyboard, or with the mouse and mouse button(s). The text from the character where you start Visual mode, up to and including the character under the cursor, is highlighted. Generally `v` and `V` modes allow definition of nonrectangular blocks, whereas `<Ctrl+V>` allows definition of only rectangular blocks.

Step 3: Type a prescribed operator command. The highlighted characters from Step 2 will be operated upon depending on the nature of the prescribed operator listed.

You can use `<Esc>` to stop the definition of a block any time before you use a prescribed operator.

A simple example that illustrates how you can copy and paste using Visual mode follows:

Step 1: At the shell prompt type **`vim visualtest1`**, then press **`<Enter>`** on the keyboard.

Step 2: Type three or four arbitrary lines of text of uneven length (five to ten words each) into the buffer that opens on screen. Put some spaces at the beginning of some of the lines.

Step 3: Position the cursor on the first character of the first line of the buffer.

Step 4: Type **v**. On the last line display you will be notified that you have entered Visual mode! Now you can define the block that will be all or possibly only a portion of Step 2 text.

Step 5: Expand the highlighted area by using the input device of your choice until all the text you typed in Step 2 is highlighted. If you make a mistake in defining the block, use **`<Esc>`** to stop the block definition and begin highlighting again at the first character in the buffer until you get the block definition you desire.

Step 6: Type **`y`**.

Step 7: On the last line display you will see a report of how many lines you just yanked.

Step 8: Position the cursor anywhere on the last line of the buffer.

Step 9. Type **`o`**. A new line opens below the last line in the buffer. Press **`<Esc>`**.

Step 10: Type **`p`**. The yanked block from Step 6 is put back in the buffer, starting on the new line you opened in Step 9 and proceeding downward. Save the file if you want to.

3.2.6.3 Using Gvim to Cut and Paste between Multiple Open Buffers

To illustrate the speed and efficiency of using gvim as a modern graphical UNIX text editor, and to describe some of gvim's functions, the following practice session allows you use gvim to create text in two different files, open buffers into those files in two different windows, and copy and paste between those buffers.

In general, all of the vi commands and operating modes shown previously work in both vim and gvim.

3.2.6.4 Practice Session 3.5

Step 1: At the shell prompt, type **`gvim gvim1`** and then press the **`<Enter>`** key.

Step 2: Type A and then type **`This is the first line of text.`** Then press the **`<Enter>`** key twice.

Step 3: Type **`This is the third line of text.`** Then press the **`<Enter>`** key.

Step 4: From the gvim pull-down menu, make the choice **`Window>Split`**. You now are looking into two windows on the same buffer.

Step 5: From the gvim pull-down menu, make the choice **`File>Save`**. Click the OK button in the `Save` window. The buffer is saved to the file **`gvim1`**.

Step 6: Use the mouse and click anywhere in the lower window with the left mouse button. You are now working in the lower buffer.

Step 7: From the gvim pull-down menu, make the choice **File>Save As**. In the `Name` box, change the name of the file to **gvim2**, and then make the **Save** button choice. The buffer is saved as **gvim2**, and you are looking into the buffer through two windows.

Step 8: The active buffer is still seen in the lower window. Use the mouse and **`<Delete>`** key on the keyboard to change the word `first` to the word `second`, and the word `third` to the word `fourth` in the lower window.

Step 9: Click anywhere in the top window.

Step 10: Make the gvim pull-down menu choice **File>Open**. Scroll down and open **gvim1** in the current directory by selecting it and making the `OK` button choice. You should now be seeing **gvim1** in the upper window, and **gvim2** in the lower window.

Step 11: Click anywhere in the bottom window.

Step 12: Use the mouse and left mouse button to highlight the text `This is the second line of text.` Make sure the cursor is flashing on the period as you finish selecting that line.

Step 13: Make the gvim pull-down menu choice **Edit>Copy**. You have "yanked" a line of text in the lower buffer graphically.

Step 14: Click on the second blank line in the upper window buffer.

Step 15: Make the gvim pull-down menu choice **Edit>Paste**. The line `This is the second line of text.` is now on the second line of the upper window buffer. You can use the gvim pull-down menu choice **Edit>Undo** to correct mistakes in copying and in pasting.

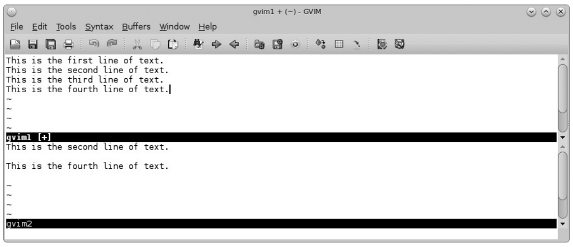
Step 16: Repeat Steps 11 through 15 to copy and paste the line `This is the fourth line of text.` from the lower window buffer to the upper window buffer. When you are done, your screen display should look similar to Figure 3.4.

Step 17: While the active window buffer is the upper window, make the gvim pull-down menu choice **File>Save-Exit**.

Step 18: At the shell prompt, type **more gvim1**. What appears on screen? Do the same for the file **gvim2**. What appears on screen?

3.2.7 Changing vi, vim, and gvim Behavior

In general, all of the environment options commands shown in this section work in vi, vim, and gvim. Note that, because vim stands for *vi improved*, vim and gvim have many more environmental options. As previously suggested, you can create an empty version of the **~/.vimrc** file to enable many of the behavioral changes we show here. We also suggest you modify both your **~/.exrc** and **~/vimrc** files to accomplish the behavioral changes



```
This is the first line of text.
This is the second line of text.
This is the third line of text.
This is the fourth line of text.
~
~
~
~
gvim1 [+]
This is the second line of text.

This is the fourth line of text.

~
~
~
~
gvim2
```

FIGURE 3.4 File **gvim1** after Step 16.

illustrated, depending on which editor you want the changes to be implemented in. If you put a *vim-specific behavior-changing option* in the **.exrc** file, when you run vi, you will probably get a warning message in vi, but not a fatal error message.

You can modify any of several environment options to customize the behavior of the vi, vim, and gvim editors, either when you are in the editor at a given time, or for every editor session. These options include, for example, specifying maximum line length and automatically wrapping the cursor to the next line, displaying line numbers as you edit a file, and displaying the mode that the editor is in. You can use full or abbreviated names for most of the options. Some of the most important and useful options and their abbreviations are summarized in Table 3.8. Also see Table 3.9 for a summary of the use of the `set` command.

The `set` command in Last Line mode changes environmental options. There are two types of environmental options that can be modified with the `set` command: toggle options, which are either "on" or "off," and options that require the use of an argument.

For example, after typing `:set showmode`, you have toggled the mode display "on," and the editor displays the current mode at the bottom of the screen. If you then type `:set noshowmode`, you have toggled the mode display "off." Similarly, after typing `:set nu`, vi displays the line numbers for all the lines in the file. To turn "off" the line number display, type `:set nonu`. When the `:set ai` command has been executed, the next line is aligned with the beginning of the previous line. This useful feature allows you to easily indent source codes that you compose with vi. Pressing `<Ctrl+D>` on a new line moves the cursor to the previous indentation level.

TABLE 3.8 Important Environmental Options for Vi, Vim, and Gvim

Option	Abbreviation	Purpose
`autoindent`	`ai`	Aligns the new line with the beginning of the previous line.
`ignorecase`	`ic`	Ignores the case of a letter during the search process (with a / or the ? command).
`list`	`list`	Displays invisible characters, such as `^I` for `<Tab>` and a $ for end-of-line characters.
`nolist`	`nolist`	Turns off the display of invisible characters.
`noignorecase`	`noic`	Instructs cases to be case sensitive.
`number`	`nu`	Displays line numbers when a file is being edited; line numbers are not saved as part of the file.
`nonumber`	`nonu`	Hides line numbers.
`scroll`		Sets the number of lines to scroll when the `<Ctrl+D>` command is used to scroll the vi screen up.
`set`		Displays all the vi variables that are set.
`all`		Displays all set vi variables and their current values.
`showmode`	`smd`	Displays the current vi mode in the bottom right corner of the screen.
`noshowmode`	`nosmd`	Turns off the mode of operation display.
`wrapmargin`	`wm`	Sets the wrap margin in terms of the number of characters from the end of the line, assuming a line length of 80 characters.

TABLE 3.9 Last Line Mode Syntax

Last line mode syntax	What it does
`abbr` command	
`:ab in out`	Use `in` as abbreviation for `out` in Insert mode.
`:unab in`	Remove abbreviation for `in`.
`:ab`	List abbreviations.
`map!, map` commands	
`:map string sequence`	Map characters string as sequence of commands. Use #1, #2, etc., for the function keys.
`:unmap string`	Remove map for characters string.
`:map`	List character strings that are mapped.
`:map! string sequence`	Map characters string to input mode sequence.
`:unmap! string`	Remove input mode map (you may need to quote the characters with `<Ctrl+V>`).
`:map!`	List character strings that are mapped for input mode.
`qx`	Record typed characters into register specified by letter x (vim and gvim).
`q`	Stop recording (vim and gvim).
`@x`	Execute the register specified by letter `x`. Use @@ to repeat the last @ command.
`set` command	
`:set x`	Enable boolean option `x`, show value of other options.
`:set nox`	Disable option `x`.
`:set x=value`	Give value to option `x`.
`:set`	Show changed options.
`:set all`	Show all options.
`:set x?`	Show value of option `x`.

To see a listing of what all environment options in the editor are (the ones you have modified and the defaults) at any time, type `:set all`.

To see a listing of what environment options you have modified, either for this session only, or for all sessions, type `:set`.

When you use `set` to modify the environment options within an editor session, the options are set for that session only!

If you want to customize your environmental options for all vi, vim, and gvim sessions, you need to put your options in the **.exrc** file in your home directory. You can use the set command to modify one or more options in the **.exrc** file as follows (typing the two keyboard keys `<Ctrl+C>` terminates the creation of the cat command):

```
$ cat > .exrc

set wm=5 showmode nu ic

<Ctrl+C>

$
```

The `wm=5` option sets the wrap margin to 5, and is an example of a set command that requires an argument. That is, each line will be up to 75 characters long. The `ic` option allows you to search for strings without regard to the case of a character. Thus, after this option has been set, the `/Hello/` command searches for strings `hello` and `Hello`.

EXERCISE 3.6

After examining Tables 3.8 and 3.9, select a few of the environment options that most appeal to you and then place them in your **~/.exrc file**. Test them by running vi.

EXERCISE 3.7

If you haven't already done so, place the `set showmode` environment setting in your **~/.exrc** and in your **~/.vimrc** file. Run vim and then gvim. Do various operations in both those editors. Does the mode you are in appear in the mode line in both editors?

3.2.8 Executing Shell Commands from within vi, vim, and gvim

At times you will want to execute a shell command without quitting vi and then restarting it. You can do so in Command mode by preceding the command with `:!`. Thus, for example, typing `:! pwd` would display the pathname of your current directory, and typing `:! ls` would display the names of all the files in your current directory. After executing a shell command, the editor returns to Command mode.

3.2.9 vi, vim, and gvim Keyboard Macros

Vi, vim, and gvim offer a variety of *macro* facilities; a macro is a keystroke construction that uses one or more compact set keystrokes to represent another larger number of keystrokes that are substituted for the single or compact set. Macros are used in vi, vim, and gvim for the following reasons:

1. During Insert mode, to construct an abbreviation. For example, in a text file where you use often-repeated blocks of the same text.
2. In Command mode, vi, vim, and gvim commands can be associated or *mapped* to other keys, such as the function keys at the top of the keyboard.
3. Complex commands and their arguments can be triggered by a single keystroke or a shorter sequence of keystrokes.

Here is a brief summary description of the various vi, vim, and gvim macro operations, two of which are covered in the indicated subsections below:

Text abbreviation (Section 3.2.9.1), which operates in Insert mode. An abbreviation works only in vi, vim, and gvim Insert mode.

Keystroke mapping (Sections 3.2.9.2 and 3.2.9.3), which can operate either in Insert mode or in Command mode, and uses the `map!` and `map` Last Line mode commands.

Once defined, a `map!` sequence is triggered only in Insert mode, and a `map` sequence is triggered only in vi, vim, and gvim Command mode.

Text-buffer execution, which operates only in Command mode. Once text has been placed in any of the named text buffers, that text can be executed as if it were a sequence of vi, vim, and gvim commands.

In the following sections, we will describe and give examples of some of the vi, vim, and gvim macro facilities, and also give an additional example of a specialized vim macro feature that can be used in gvim as well. Table 3.9 summarizes the uses of the `abbr`, `map!`, and `map` commands.

3.2.9.1 Text Abbreviation Macros Used in Insert Mode

To save keystrokes while entering text, in Last Line mode, use the `abbr`(eviate) or just `ab` command.

It has the following general syntactic form:

```
:ab[br] [abbreviation abbreviated]
```

where:

`:` gets you into Last Line mode;

`[ ]` designates optional components;

`ab` or **`abbr`** is the command for creating an abbreviation;

`abbreviation` is a valid string of contiguous (no spaces allowed) characters; and

`abbreviated` is the substitute text you want to be placed in the buffer.

Text abbreviations can be canceled with the Last Line mode `unabbr` command, followed by typing the abbreviation you want to cancel. Also, if you just type `abbr` in Last Line mode, you get a listing of all the abbreviations that are active.

To use the abbreviation, <u>when you are in Insert mode</u>, whenever you type the string that represents `abbreviation` and precede as well as follow it by a nonalphanumeric character, the substitution will take place.

The editor will examine the character before and the next character after you type the `abbreviation` to see if it's nonalphanumeric or underscore, and if so, `abbreviation` will be erased and the string that represents `abbreviated` will be substituted for it. Also, you are no longer in Insert mode.

For example, in Last Line mode, if you type `ab kts Know this stuff!` and then press `<Enter>`, `kts` is the abbreviation. Then anywhere in Insert mode, when you type `sts` and precede it and follow it by pressing the space key, the left or right arrow keys (all of which yield nonalphanumeric characters and are not the underscore keys on the keyboard for our system), the string `Know this stuff!` will be substituted on that line, and you will no longer be in Insert mode.

Note: With `abbr`, your text appears as you type it, and no substitution is performed on `abbreviation` until you type nonalphanumeric characters before it and after it. As shown in the next section, this is different from keystroke mapping using the `map!` or `map` commands.

The following are some useful abbreviations for Python program file creation.

```
:ab 1 #!/usr/local/bin/python

:ab 2 from Tkinter import *

:ab 3 import os

:ab 4 import sys
```

3.2.9.2 Keystroke-Mapping Macros Used in Insert Mode

`map`, shown in the next subsection, works on characters that are typed in Command mode, and `map!`, shown in this subsection, works on characters that are typed in Insert mode.

As shown in the previous section, `abbr` won't substitute text until you type a nonalphanumeric before and after the `abbreviation` string. Notice the editor echoes each character of the `abbreviation` as you type it, just in case you really want the string of characters that represents `abbreviation` to be an actual string of characters that you want in your text. Keystroke mapping works in a more keystroke- and time-dependent way. Keystroke mapping used in Insert mode is handled by the Last Line mode `map!` command, which takes the following general form:

```
:map! [substitution substituted]
```

where:

: gets you into Last Line mode;

[ ] designates optional components;

map is the command for creating a keyboard mapping;

substitution is a valid string of contiguous (no spaces allowed) characters; and

substituted is the substitute text you want to be placed in the buffer.

For example, in Last Line mode, if you type `map! ts This will save you time!` and then press `<Enter>`, `ts` is the `substitution`. Then, anywhere in Insert mode, when you type `ts` in a short amount of time (under approximately half a second), the string `This will save you time!` will be substituted on that line <u>and you will still be in Insert mode</u>. If you type more slowly, the literal string `ts` will be inserted.

The keystroke sequence `<Ctrl+V>` will let you escape the mapping, as long as you precede the macro with it. So no matter how fast you type in `<Ctrl+V> ts`, you get the literal string `ts` inserted.

Remapping abbreviations can be canceled with the Last Line mode `unmap!` command, followed by typing the `substitution` you want to cancel. Also, if you just type `map!` in Last Line mode, you get a listing of all the mappings that are active. You will see that the editor already has several mappings defined by default.

3.2.9.3 Keystroke-Remapping Macros Used in Command Mode

Command mode remapping is accomplished with the `map` Last Line mode command.

The general form of the `map` command is as follows.

```
:map [substitution substituted]
```

where:

`:` gets you into Last Line mode;

`[ ]` designates optional components;

`map` is the command for creating a keyboard mapping;

`substitution` is a valid string of contiguous(no spaces allowed) characters; and

`substituted` is the substitute text you want to be placed in the buffer.

Some editor command keys cannot be remapped in Command mode. Two examples of these keys are `:` (colon) and `u`.

Remapping substitutions can be canceled with the Last Line mode `unmap` command, followed by typing the remapping you want to cancel. Also, if you just type `map` in Last Line mode, you get a listing of all the mappings that are active. You will see that the editor already has several mappings defined by default.

As an example, in Last Line mode, if you type `map #8: wq<Ctrl+V><Ctrl+Enter>`and then press `<Enter>`, the function key `F8>` at the top of your keyboard is the `substitution`. The `substituted` is the command to write the buffer to a file and quit the editor. The `<Ctrl+V>` and `<Ctrl+Enter>` keystrokes are the way to enter control characters on the command line, in this case the `<Enter>` key at the end of the command. After this mapping is done, anytime you are in Command mode, when you press the function key `<F8>`, the buffer will be written to the default file and you will exit the editor.

Another interesting and useful example is the following `map` command, which can be placed in your **.exrc** file, so that you can use the function key `<F3>` during editor sessions to generate a skeleton C program construct:

```
map #3 ^[i#include stdio.h ^Mmain(argc, argv) ^M int argc;^M
  char #argv[];^Ml{^M}^M^[
```

where:

^[stands for pressing `<Ctrl+V>` and then `<Esc>`; and

^M stands for pressing `<Ctrl+V>` and then the `<Enter>` key

The relative number of spaces in this `map` command definition controls the indentation of the skeleton construct. Also, the `^M` entries put each of the skeleton construct components on a new line.

3.2.9.4 Vim/Gvim Macro Example

Here is a repeat of Practice Session 3.4, slightly enlarged, that uses a vim-specific macro command sequence to accomplish the same thing that Practice Session 3.4 did, but in another way.

3.2.9.5 Practice Session 3.6

Step 1: From the shell prompt, type **`vim unixos2`** and then press **`<Enter>`** on the keyboard.

Step 2: In vim, type **`A`** and then type the following 10 lines of text, each on its own line:

```
computer scientists
students
hackers
systems analysts
newbies
UNIX gurus
computer programmers
systems administrators
network administrators
LINUX users
```

Step 3: Press **`<Esc>`**, then place the cursor anywhere on the first line of text.

Step 4: Type **`q a`**. This puts you in record mode and associates the macro you are about to record with the **`a`** key.

Step 5: Type **`I`**. The cursor is now at the start of the first line in Insert mode.

Step 6: Type **`UNIX is the operating system of choice for`** with a single space after the **`r`** in the word **`for`**. Press **`<Esc>`**.

Step 7: Place the cursor anywhere on the second line of text.

Step 8: Type **q**. This ends record mode.

Step 9: Type **a@9**. This "plays back" the macro defined with the **a** key nine times, once on each of the lines below the first line, inserting the text string `UNIX is the operating system of choice for`.

Step 10: Save the file, print it out, and memorize its contents.

3.3 THE EMACS EDITOR

The emacs editor is the most complex and customizable of the UNIX text editors, and it gives you the most freedom, flexibility, and control over the way you edit text files. It can format text for very specific technical applications, such as program source code development, more effectively than a word processor. Its use in that application makes the process of program development more efficient. In addition, from within the emacs program, in multiple windows, you can accomplish a wide variety of personal productivity and operating system tasks, such as sending e-mail and executing shell commands and scripts. But along with more control, specificity, and capabilities comes some additional learning in terms of a more complex keystroke command structure. This complexity can be offset in part for some users, and totally for others, by using the graphical forms of input and command execution that we will emphasize in the sections that follow.

To stress how the keyboard keys are used in GNU emacs, we repeat the following note shown at the beginning of this chapter here:

1. Pressing the Escape key is signified as `<Esc>`
2. Pressing the Enter key is signified as `<Enter>`
3. Pressing the `<Ctrl>` key in combination with another single key is signified as `<Ctrl+X>`, where you hold down the `<Ctrl>` key and press the `X` key (or any valid key for that combination) at the same time.
4. Pressing the `<Alt>` key in combination with another single key is signified as `<Alt+X>`,

   where you hold down the `<Alt>` key and press the `X` key (or any valid key for that combination) at the same time.
5. A variant of 3. and 4. is shown as `<Ctrl+X> a [b]`, where you first press and release `<Ctrl>` and `X` simultaneously, and then press the `a` key, and optionally press the `b` key (or any valid combination of single keys or strings of characters.
6. In GNU emacs for PC-BSD and Solaris, the `Meta` key that is referred to in much of the literature on GNU emacs, is the `<Alt>` key.

It is important to realize before you begin that there are some common terms used in vi, vim, and gvim (the editors from the previous section) and emacs that describe the facilities of each editor, but the terms do not have the same meaning between the two major families of editor.

As you saw in Section 3.1.2, with vi, vim, and gvim, you can't immediately begin to enter text into the file you are editing. You have to be in Insert mode to do that, that's what typing `A` as the second step is doing. Vi, vim, and gvim have modes. In GNU emacs, you can start typing text into the file immediately. Emacs is a *modeless* editor.

Vi, vim, and gvim operate in three distinct modes: Command mode, Insert mode, and Last Line mode. Emacs is a *modeless* editor in the sense that, when you launch emacs, you do not have to switch modes to immediately type characters on the keyboard and enter text into a buffer, or change modes to save the buffer to a file.

For example, emacs does have major modes of operation, such as *Lisp mode*, *Python mode*, and *C mode*, but they are for the special formatting of text and for specialized operations when editing files for use in those language applications. This is different from allowing you to switch between significant forms of action in the editor, as the vi, vim, and gvim Command, Insert, and Last Line modes do, as seen in the previous section. The keystroke command syntax itself in emacs is different and more complex than in vi, involving use of the `<Ctrl>` and `<Alt>` prefix characters, as previously noted. The emacs concepts of *point* and the cursor location are also more refined and specific than in vi. In emacs, the point is the location in the buffer where you are currently doing your editing; the point is assumed to be at the left edge of the cursor, or always between characters or white space (what you enter into a text file when you press the space bar). This difference becomes an important issue when you want to use the cut/copy/paste operations. In vi, yanking removes text from the main buffer, much like cutting/copying, whereas in emacs yanking is more like pasting into the main buffer. The concept of a buffer is very important in emacs, and is very much the same in emacs as it is in vi.

Currently, there is one major "brand" of emacs for UNIX: GNU emacs. We use the graphical form of GNU emacs version 24.3.1 in both PC-BSD and Solaris, running in its own frame, in the following illustrations, exercises, practice sessions, and problems.

If you cannot run a graphical emacs because you are working in a text-only console or terminal, you can still gain access to the Menu Bar at the top of the emacs screen by pressing `<Esc>` on the keyboard and then pressing the single back quote (`) key. You can then descend through the menu bar choices by pressing the letter key of the menu choice you want to make. For example, pressing the `f` key gives you access to the File pull-down menu choices, and then pressing the `s` key allows you to save the current buffer. Unfortunately, you cannot access the speed button bar menu choices from within a text-only display of emacs. A summary of emacs commands is given in Table 3.10.

3.3.1 Launching Emacs, Emacs Screen Display, General Emacs Concepts and Features

The general syntax for launching the emacs program from the command line in a console window is as follows (anything enclosed in square brackets `[ ]` is optional):

TABLE 3.10 Summary of emacs Commands

Command	Action
`<Ctrl+X> <Ctrl+F>`	Visit a file (`find-file`)
`<Ctrl+X> <Ctrl+R>`	Visit a file for viewing, without allowing changes to it (`find-file-read-only`)
`<Ctrl+X> <Ctrl+V>`	Visit a different file instead of the one visited last (`find-alternate-file`)
`<Ctrl+X> 4 f`	Visit a file, in another window (`find-file-other-window`)
`<Ctrl+X> 5 f`	Visit a file, in a new frame (`find-file-other-frame`)
`<Alt+X> find-file-literally`	Visit a file with no conversion of the contents (shows you control characters, etc.)
`<Ctrl+X> <Ctrl+S>`	Save the current buffer to its file (`save-buffer`)
`<Ctrl+X> s`	Save any or all buffers to their files (`save-some-buffers`)
`<Alt+~>`	Forget that the current buffer has been changed (`not-modified`)
`<Ctrl+X> <Ctrl+W>`	Save the current buffer with a specified file name (`write-file`)
`<Alt+X> set-visited-file-name`	Change the file name under which the current buffer will be saved
`<Ctrl+R>`	View the buffer that you are currently being asked about
`<Ctrl+H>`	Display a help message about these options
`<Ctrl+X> <Ctrl+C>`	Exits emacs
`<Ctrl+X> <Ctrl+Z>`	Suspends emacs and exits to the shell

SYNTAX

`emacs [options][file(s)]`

Purpose: Allows you to edit a new or existing file(s)

Output: With no options or file(s) specified, emacs runs and begins or opens on the Welcome Screen buffer

Commonly used options/features:

`+n`	Begin to edit file(s) starting at line number **n**
`-nw`	Run emacs without opening a window, useful in an elementary GUI environment
`emacs file1 file2 file3`	Open three buffers in emacs on three different files at the same time

For example, if you run the emacs program by typing `emacs alien` in a terminal window in our base PC-BSD system, emacs launches and shows the Welcome Screen buffer.

To close the Welcome Screen buffer display that opens in the bottom window of the emacs frame, while the cursor is flashing in the top window, make the emacs pull-down menu choice **Remove Other Windows**. Or you can type `<Ctrl+X> 1`. In either case, you will only have one buffer shown in the screen display, similar to Figure 3.5.

A brief description of the major components of the emacs screen display labeled in Figure 3.5 is as follows (note: items J, A, B, D, and C are found on what is called the *mode line*):

A. Name of the current buffer: This is the name of the entity or "file" you are editing in this window. In Figure 3.5, the name of the buffer is **alien**.

B. Major and minor mode: Different major modes are used to edit different kinds of files, like C programs, Lisp, or HTML, and special configurations of the major modes define the minor modes. In Figure 3.5, only the major mode `Fundamental` is shown, with no minor mode set.

C. Percentage of the text shown on screen: This shows how much of the text in the buffer is seen on screen. In Figure 3.5, `All` of the text in the current buffer is shown on screen.

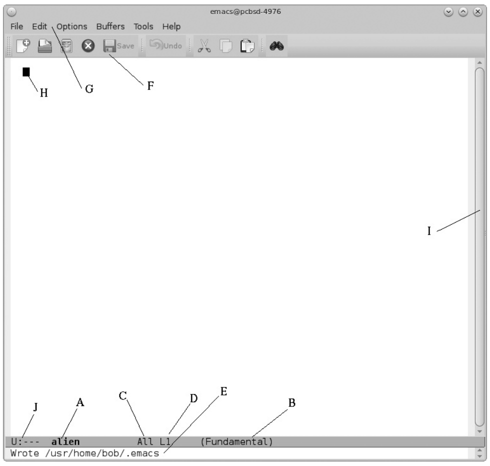
D. Current line number: The line location of the cursor in the current buffer is displayed here.

E. Minibuffer: Information and questions/prompts from emacs appear here. In Figure 3.5, `Wrote /usr/home/bob .emacs` is shown on screen, because the help file initial screen display buffer was closed.

F. Speed button bar: This allows you to do quick, common operations graphically.

G. Menu bar: This gives you pull-down menus that contain all of the important emacs operations.

H. Text: The actual text you are editing appears here.



```
U:---  alien          All L1     (Fundamental)
Wrote /usr/home/bob/.emacs
```

FIGURE 3.5 First GNU emacs screen display.

I. Scroll bar: The scroll bar allows you to graphically scroll or move through the text.

J. Status indicator: Two-character codes are used to tell you about your file. In Figure 3.5, a `U-` and two hyphens (`U---`) indicate that the file has not changed in emacs and is the same as the version saved to disk, and that you can work on the file.

3.3.1.1 Emacs Help

Emacs provides a wide variety of help commands, all accessible through the key sequence `<Ctrl+H>` or graphically with the function key `<F1>`. You can also type `<Ctrl+H>` `<Ctrl+H>` to view a list of help commands. You can scroll the list with `<Space>` and `<Del>`, then type the help command you want. To cancel, type `<Ctrl+G>`. Many help commands display their information in a special help buffer. In this buffer, you can type `<Space>` and `<Del>` to scroll and press `<Enter>` to follow hyperlinks.

The following are the most general ways of obtaining help on a topic or command:

```
<Ctrl+H> a topic(s) <Enter>
```

This searches for commands whose names match the argument **`topic(s)`**. The argument can be a keyword, a list of keywords, or a regular expression.

```
<Ctrl+H> i d m emacs <Enter> i topic <Enter>
```

This searches for **topic** in the indices of the emacs Info manual, displaying the first match found. Press **,** (comma) to see subsequent matches. You can use a regular expression as a topic.

```
<Ctrl+H> i d m emacs <Enter> s topic <Enter>
```

Similar, but searches the text of the manual rather than the indices.

```
<Ctrl+H> <Ctrl+F>
```

This displays the emacs FAQ, using Info.

```
<Ctrl+H> p
```

This displays the available emacs packages based on keywords.

A summary of help command syntax is found in Table 3.11.

3.3.1.2 Graphical Features

The most useful graphical features of emacs are the menu bar and speed button bar seen in Figure 3.5 as F and G. These features incorporate all of emacs's functionality into a

TABLE 3.11 Summary of Help Command Syntax

`<Ctrl+H> a topics <Enter>`	Display a list of commands whose names match `topics` (`apropos-command`).
`<Ctrl+H> b`	Display all active key bindings—minor mode bindings first, then those of the major mode, then global bindings (`describe-bindings`).
`<Ctrl+H> c key`	Show the name of the command that the key sequence `key` is bound to (`describe-key-briefly`). Here `c` stands for "character." For more extensive information on `key`, use <Ctrl+H> `k`.
`<Ctrl+H> d topics <Enter>`	Display the commands and variables whose documentation matches topics (`apropos-documentation`).
`<Ctrl+H> e`	Display the ***Messages*** buffer (`view-echo-area-messages`).
`<Ctrl+H> f function press <Enter>`	Display documentation on the Lisp function named `function` (`describe-function`). Since commands are Lisp functions, this works for commands too.
`<Ctrl+H> h`	Display the **HELLO** file, which shows examples of various character sets.
`<Ctrl+H> i`	Run Info, the GNU documentation browser (`info`). The emacs manual is available in Info.
`<Ctrl+H> k key`	Display the name and documentation of the command that `key` runs (`describe-key`).
`<Ctrl+H> l`	Display a description of your last 300 keystrokes (`view-lossage`).
`<Ctrl+H> m`	Display documentation of the current major mode (`describe-mode`).
`<Ctrl+H> n`	Display news of recent emacs changes (`view-emacs-news`).
`<Ctrl+H> p`	Find packages by topic keyword (`finder-by-keyword`). This lists packages using a package menu buffer.
`<Ctrl+H> P package <Enter>`	Display documentation about the package named `package` (`describe-package`).
`<Ctrl+H> r`	Display the emacs manual in Info (`info-emacs-manual`).
`<Ctrl+H> s`	Display the contents of the current syntax table (`describe-syntax`). The syntax table says which characters are opening delimiters, which are parts of words, and so on.
`<Ctrl+H> t`	Enter the emacs interactive tutorial (`help-with-tutorial`).
`<Ctrl+H> v var <Enter>`	Display the documentation of the Lisp variable var (`describe-variable`).
`<Ctrl+H> w command <Enter>`	Show which keys run the command named command (`where-is`).
`<Ctrl+H> C coding <Enter>`	Describe the coding system `coding` (`describe-coding-system`).
`<Ctrl+H> C <Enter>`	Describe the coding systems currently in use.
`<Ctrl+H> F command <Enter>`	Enter Info and go to the node that documents the emacs command `command` (`Info-goto-emacs-command-node`).
`<Ctrl+H> I method <Enter>`	Describe the input method `method` (`describe-input-method`).
`<Ctrl+H> K key`	Enter Info and go to the node that documents the key sequence `key` (`Info-goto-emacs-key-command-node`).
`<Ctrl+H> L language-env <Enter>`	Display information on the character sets, coding systems, and input methods used in language environment `language-env` (`describe-language-environment`).
`<Ctrl+H> S symbol <Enter>`	Display the Info documentation on symbol `symbol` according to the programming language you are editing (`info-lookup-symbol`).
`<Ctrl+H>`	Display the help message for a special text area, if the point is in one (`displaylocal-help`). (These include, for example, links in ***Help*** buffers.)

graphical style of interaction. When a menu choice is grayed out, that means it is not available at the current level you are operating at. The following is a brief description of what tasks each menu bar item accomplishes.

File: Facilities for opening, saving, and closing buffers, files, windows, and frames

Edit: Means to modify text in buffers

Options: Facilities to make configuration changes

Buffers: A pull-down menu listing of the currently open buffers

Tools: File and application functions

Help: Extensive documentation and online manual for emacs

The speed button bar contains single-button presses for (1) file and buffer operations; (2) common text-editing operations, such as cut and paste; and (3) printing, searching, and changing preferences.

3.3.1.2.1 Buffers, File, Windows, and Frames The most important concept in emacs is that of a buffer, or text object that is currently being edited by emacs. This is different from a file, which is a text object stored on disk. The differentiation is made, in simple terms, because (1) the object currently being modified and viewed in emacs cannot be the same object stored on disk as you have not yet saved your edits; and (2) emacs can work on text objects that are not files and never will be, such as the output from commands typed on the UNIX command line. When you first launch emacs and specify a file to edit, you are looking into the buffer created by emacs for that file in what is generally known as an emacs frame, with a single window open to allow you to see the buffer contents. A frame consists of one window, or possibly many windows, tiled in it, the pull-down and speed button bar menus, the mode line, and a minibuffer. In Practice Sessions 3.12, 3.13, and 3.14, you will work with multiple buffers viewed in emacs using multiple windows in one frame only.

3.3.1.2.2 Point, Mark, and Region The second most important concept in emacs is that of the *point* and *mark*, and the *region* of text they demarcate. The point is located in the white space before the character the cursor is highlighting. The mark, set by placing the cursor over a character and then holding down `<Ctrl+Space>` or `<Ctrl+@>`, is also in the white space before the character the cursor is highlighting. The region or area of text you want to manipulate in operations such as cutting and copying, is all text between the point and the mark. For example, in the line of text `Now is the time for all good men`, if the cursor is on or highlighting the `N` in the word `Now` (the point is in the white space before the `N`), and the mark has been set before the character `i` in the word `time` by placing the cursor on the letter `i` and holding down `<Ctrl+Space>`, then the region is defined as `Now is the t`.

To exit from emacs without saving any of the buffers, make the pull-down menu choice **File>Quit** or type `<Ctrl+X>/<Ctrl+C>` on the keyboard.

3.3.2 How to Use Emacs to Do Shell Script File Creation, Editing, and Execution

The following practice session shows how to create a file to define aliases, or command name substitutes, that allow you to type DOS command names at the UNIX shell prompt to execute some of the common UNIX file maintenance operations. DOS commands are similar to UNIX commands but are used in the Windows operating system environment. As you will see in the practice session, you can use an efficient combination of keyboard typing and graphical interaction to work with emacs.

In this section, we assume that you are running the C shell, which is the default shell on our PC-BSD system. On Solaris the default shell is the Bourne Again shell, commonly referred to as *Bash*. So in the following steps, substitute the correct syntax for the Bash alias commands for the C shell alias commands we show. Also, it is assumed in these practice sessions, exercises, and problems that you are creating and editing files in your home directory.

3.3.2.1 Practice Session 3.7

Step 1: At the shell prompt, type **`emacs alien`** and then press **`<Enter>`**.

The emacs screen appears in your display, similar to Figure 3.5. Close the Welcome Screen as noted in Section 3.3.1 by holding down **`<Ctrl+X>`** and then pressing the **`1`** numeric key.

Step 2: Type **`# DOS aliases`** and then press **`<Enter>`**.

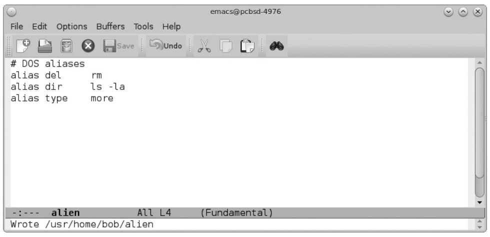
Step 3: Type **`alias del rm`** and then press **`<Enter>`**.

Step 4: Type **`alias dir ls -la`** and then press **`<Enter>`**.

Step 5: Type **`alias type more`** and then press **`<Enter>`**.

Step 6: Hold down the **`<Ctrl+X>`**, and then hold down **`<Ctrl+S>`** to save your file with the name **alien**. The display of your text should appear similar to Figure 3.6.

Step 7: Hold down **`<Ctrl+X>`**, and then hold down **`<Ctrl+C>`** to gracefully exit from emacs and return to the C shell prompt.



```
# DOS aliases
alias del     rm
alias dir     ls -la
alias type    more
```

```
-:---  alien         All L4      (Fundamental)
Wrote /usr/home/bob/alien
```

FIGURE 3.6 File **alien** after Step 6.

3.3.3 Visiting Files, Saving Files, and Exiting

UNIX stores data permanently in files, so most of the text you will edit with emacs comes from a file and is saved in a file. To edit a file while running emacs, you need to read the file into a buffer and prepare that buffer containing a copy of the file's text. This is called *visiting* the file. (Note: The emacs editing commands work on the text in the buffer inside emacs. Your changes are written to the file itself only when you save the buffer to the file.)

In addition to visiting and saving files, emacs can delete, copy, rename, and append to files, keep multiple versions of them, and operate on file directories.

The following are some of the basic operations you can do to visit files, save them, and then exit gracefully from emacs:

* Visiting a New File

To visit a new file from within emacs, make the pull-down menu choice **File>Visit New File**. In the `Name` bar that appears in the `Find File` window on screen, type in a new file name and then make the choice `OK`. If the only buffer open is the welcome window, it will close, and you will be editing a buffer named with the new file name.

You can do the same thing by typing `<Ctrl+X> <Ctrl+F>`. Then in the minibuffer, type in the file name.

* Saving to a File without Quitting Emacs

After you have entered some text into the current buffer, make the pull-down menu choice **File> Save**.

You can do the same thing by typing `<Ctrl+X> <Ctrl+S>`.

* Saving to a File with Unsaved Changes and Quitting Emacs:

If you make unsaved changes to a buffer, and make the pull-down menu choice **File>Quit**, emacs puts a `Question` dialog box on screen asking you the following:

```
Save file ? Yes No

View This Buffer

View Changes in This Buffer

Save This but No More

No for All
```

You can do any of these, depending on what you want to accomplish. You get the same choices, although they are less descriptive, when you type `<Ctrl+X> <Ctrl+C>` for the unsaved buffer.

3.3.4 Cursor Movement and Editing Commands

In addition to general purpose commands, emacs has some important cursor movement and editing commands that allow you to move quickly and easily around the text and make changes. Some of the most important of these commands are listed in Tables 3.12 and 3.13.

Practice Session 3.8 illustrates the use of a mixture of keystroke commands and graphical methods in emacs, and lets you edit the file **alien** that you created in Practice Session 3.7,

TABLE 3.12 Entities to Move Over

Entity to Move Over		**Backward**	**Forward**
Character		`<Ctrl+B>`	`<Ctrl+F>`
Word		`<Alt+B>`	`<Alt+F>`
Line		`<Ctrl+P>`	`<Ctrl+N>`
Go to line beginning (or end)		`<Ctrl+A>`	`<Ctrl+E>`
Sentence		`<Alt+A>`	`<Alt+E>`
Paragraph		`<Alt+{>`	`<Alt+}>`
Page		`<Ctrl+X> [`	`<Ctrl+X> ]`
Sexp		`<Ctrl+Alt+B>`	`<Ctrl+Alt+F>`
Function		`<Ctrl+Alt+A>`	`<Ctrl+Alt+E>`
Go to buffer start (or end)		`<Alt+<>`	`<Alt+>>`
Scroll to next screen	`<Ctrl+V>`		
Scroll to previous screen	`<Alt+V>`		
Scroll left	`<Ctrl+X> <`		
Scroll right	`<Ctrl+X> >`		
Scroll current line to center, top, bottom	`<Ctrl+L>`		
Go to line	`<Alt+G> g`		
Back to indentation	`<Alt+M>`		

TABLE 3.13 Entities to Kill

Entity to Kill		**Backward**	**Forward**
Character (delete, not kill)		`<Del>`	`<Ctrl+D>`
Word		`<Alt+Del>`	`<Alt+D>`
Line (to end of)		`<Alt+0><Ctrl+K>`	`<Ctrl+K>`
Sentence		`<Ctrl+X> DEL`	`<Alt+K>`
Sexp		`<Alt+-> <Ctrl+Alt+K>`	`<Ctrl+Alt+K>`
Kill region	`<Ctrl+W>`		
Copy region to kill ring	`<Alt+W>`		
Kill through next occurrence of char	`<Alt+Z> char`		
Yank back last thing killed	`<Ctrl+Y>`		
Replace last yank with previous kill	`<Alt+Y>`		

so that it may be used as an *alias* for commands in the C shell. In Practice Session 3.8, we ask you to insert the file you created in Practice Session 3.7 into your home directory **.cshrc** file, so that, upon subsequent logins, these DOS-aliased commands will be available. Before you begin Practice Session 3.8, do the following (if you are using Solaris, do the following practice session and its preparation for Bash using the correct syntax of the Bash alias commands; also use the Bash configuration file **.bashrc** in your home directory, in place of the C shell **.cshrc** configuration file shown):

1. Use the `ls -la` command to find out if you have a **.cshrc** file in your home directory. If you do not, then use emacs to create a new file named **.cshrc** with nothing in it.
2. Find out which shell you are currently using by typing `echo $SHELL`. On our system, PC-BSD, the C shell is the default shell. If you are using Solaris, the default shell is Bash.

If you make a mistake anywhere in the following exercise, you can revert to using the graphical form of editing using the mouse and pull-down menus, including undo, inside the emacs window for expediency.

3.3.4.1 Practice Session 3.8

Step 1: At the shell prompt, type **`emacs alien`** and then press **`<Enter>`**. Close the Welcome Screen as noted in Section 3.3.1 by holding down **`<Ctrl+X>`** and then pressing the **`1`** numeric key. The file you created in Practice Session 3.7 is loaded into the buffer, and your screen display should look similar to the one shown in Figure 3.6.

Step 2: Using the arrow keys, position the cursor to the right of the **`"`** character at the end of the third line.

Step 3: Press **`<Enter>`**

Step 4: Type **`alice dir/w ls`**

Step 5: Hold down **`<Ctrl+A>`**. The cursor moves to the beginning of the line.

Step 6: Hold down **`<Esc+D>`**. The word `alice` has been cut from the buffer.

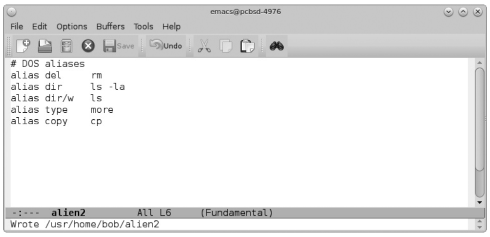
Step 7: Type **`alias`**.

Step 8: Hold down **`<Alt+B>`**. The cursor moves to the beginning of the word `alias`.

Step 9: Position the cursor with the arrow keys on the keyboard at the beginning of the first blank line, below the line that reads `alias` type **`more`**.

Step 10: Hold down **`<Ctrl+Y>`**. The cut word `alice` has been put back into the buffer at the start of the line.

Step 11: Use the arrow keys to position the cursor in the space to the right of the word `alice` if it is not there already.



```
# DOS aliases
alias del      rm
alias dir      ls -la
alias dir/w    ls
alias type     more
alias copy     cp
```

```
-:---  alien2        All L6     (Fundamental)
Wrote /usr/home/bob/alien2
```

FIGURE 3.7 Display after Step 15.

Step 12: Use the **`<Delete>`** or **`<Backspace>`** keys to delete the letters `c` and `e` from the word `alice`.

Step 13: Type **`as copy cp`**.

Step 14: Hold down **`<Ctrl+X>`** **`<Ctrl+W>`**.

Step 15: At the `Write file:` prompt, erase anything on the line with the **`<Backspace>`** key, and type **`alien2`** and then press **`<Enter>`**. Your screen display should now look similar to the one shown in Figure 3.7.

Step 16: Hold down **`<Ctrl+H>`** and then press the **`a`** key. The minibuffer area shows a prompt for you to obtain help. Hold down **`<Ctrl+G>`**. Doing so cancels your help request.

Step 17: Hold down **`<Ctrl+X>`** **`<Ctrl+C>`** to quit emacs and return to the shell prompt.

Step 18: From the shell prompt, type **`emacs .cshrc`** and then press **`<Enter>`**. The contents of your **`.cshrc`** file now appear in the editing buffer.

Step 19: Position the cursor with the arrow keys on the keyboard on any blank line in the file. Hold down **`<Ctrl+X>`** and then press the **`i`** key on the keyboard. This will allow you to insert the contents of a file into the current buffer at the position of the cursor.

Step 20: In the minibuffer, type **`alien2`**. The lines of text from **alien2**'s DOS aliases should now be inserted into the file **.cshrc** after and below where you positioned the cursor in Step 19.

Step 21: From the pull-down menu File, make the choice **File>Save (current buffer)**, or use **`<Ctrl+X>`** **`<Ctrl+S>`**.

Step 22: Hold down **`<Ctrl+X>`** **`<Ctrl+C>`** to quit emacs and return to the shell prompt.

Step 23: To test your new **.cshrc** file, log out of the current session using the desktop logout procedures and then log back in to your UNIX system again.

Step 24: In a terminal or console window, at the shell prompt, type one of the aliased commands, with its appropriate arguments if necessary, and note the results.

3.3.5 Keystroke Macros

The emacs text editor contains a simple-to-use facility that allows you to define keystroke macros, or collections of keystrokes that can be recorded and then played back at any time. This capability allows you to define repetitive multiple keystroke operations as a single command and then execute that command—as many times as you want. The keystrokes can include emacs commands and a series of keystrokes. A macro can also be saved with a name, or even be saved to a file, for use during subsequent emacs editing sessions. Table 3.15 shows a list of some of the most important keyboard macro commands.

For a more complete description and detailed explanation of how to record, edit, list, and delete keystroke macros, see Section 3.3.13.5.

Practice Session 3.9 lets you create a new text file using some of the commands presented in Table 3.15.

3.3.5.1 Practice Session 3.9

Step 1: At the shell prompt, type **`emacs datafile`** and then press **`<Enter>`**.

The emacs screen appears on your display.

Step 2: Hold down **`<Ctrl+X> <Shift+9>`**. These actions begin your keyboard macro definition. If you make a mistake anywhere in subsequent steps, simply hold down **`<Ctrl+G>`** to cancel the current macro definition.

Step 3: Type **`1 2 3 4 5 6 7 8 9 10`** and then press **`<Enter>`**.

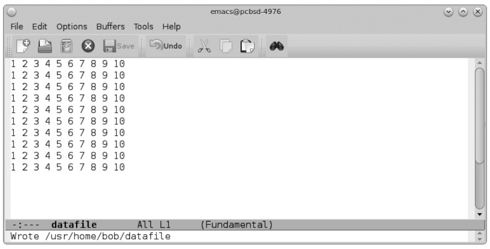
Step 4: Hold down **`<Ctrl+X> <Shift+0>`**. These actions end your macro definition.

Step 5: Hold down **`<Ctrl+X> e`**. Doing so replays the macro that you just defined, placing another line of the numbers 1 through 10 in the buffer.

Step 6: Repeat Step 5 eight more times so that your display looks similar to that shown in Figure 3.8.

Step 7: Hold down **`<Ctrl+X> <Ctrl+S>`**. These actions save the buffer to the file datafile.

Step 8: Hold down **`<Ctrl+X> <Ctrl+C>`** to exit from emacs.



```
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10

-:---  datafile       All L1     (Fundamental)
Wrote /usr/home/bob/datafile
```

FIGURE 3.8 Display after Step 6.

3.3.6 Cut or Copy and Paste and Search and Replace

As we mentioned previously, every word processor has the capability to cut or copy text and then paste that text back into the document and to search for old text and replace it with new text. Because emacs operations can be totally text activated, whereby you use sequences of keystrokes to execute commands, cutting or copying and pasting are fairly complex operations. They are accomplished with the *Kill Ring*, whereby text is held in a buffer by killing it and is then restored to the document at the desired position by yanking it. Global search and replace are somewhat less complex and are accomplished by either an unconditional replacement or an interactive replacement.

The mark is simply a place holder in the buffer. For example, to cut three words from a document and then paste them back at another position, move the point before the first word you want to cut and press `<Esc+D>` three times. The three words are then cut to the Kill Ring. Because the Kill Ring is a FIFO buffer, you can now move the point to where you want to restore the three words and press `<Ctrl+Y>`. The three words are yanked into the document in the same order, left to right, that they were cut from the document.

To copy three words of text and then paste them back at another position, set the mark by positioning the point after the three words, and then press `<Ctrl+@>` at that position. Then reposition the point before the three words; you have now defined a region between the point and the mark. There is only one mark in the document. Press `<Esc+W>` to send the text between the point and the mark to the Kill Ring; the text is sent, but it is not blanked from the screen display. To restore the three words at another position, move the point there and press `<Ctrl+Y>`. The three words are restored at the new position. Table 3.13 gives the important kill and yank commands for emacs.

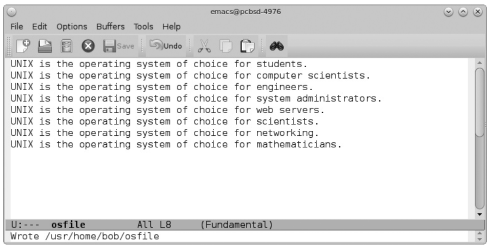
Global search and replace can be either unconditional, where every occurrence of old text you want to replace with new text is replaced without prompting, or it can be interactive, where you are prompted by emacs before each occurrence of old text is replaced with new text. Also, the grammar of replacement can include regular expressions, which we do not cover here.

For example, to replace the word `men` unconditionally with the word `women` from the current position of the point to the end of the document, press `<Esc+X>`, type `replace-string`, and then press `<Enter>`. You are then prompted for the old string. Type `men` and then press `<Enter>`. You are then prompted for the new string. Type `women` and then press `<Enter>`. All occurrences are replaced with no further prompts.

To accomplish an interactive replacement, simply press `<Esc+X>`, type `query-replace`, and then press `<Enter>`. You can then input old and new strings, but you are given an opportunity at each occurrence of the old string to replace it or not to replace it with the new string. Table 3.14 shows the actions that you can take while in the midst of an interactive search and replace. Practice Session 3.10 contains further examples of copying and pasting and global search and replace, both unconditional and interactive. Your objective will be to type in one line of text, copy it into the Kill Ring, and then paste it into the document seven times. Then modify the contents of the original line and each pasted line by using both interactive search and replace and unconditional search and replace. Upon completion of Practice Session 3.10, your screen display should look similar to Figure 3.9.

TABLE 3.14 Interactive Search and Replace

Search and Replace Action	Keystrokes
Search forward	`<Ctrl+S>`
Search backward	`<Ctrl+R>`
Regular expression search	`<Ctrl+Alt+S>`
Reverse regular expression search	`<Ctrl+Alt+R>`
Select previous search string	`<Alt+P>`
Select next later search string	`<Alt+N>`
Exit incremental search	`<Enter>`
Undo effect of last character	`<Del>`
Abort current search	`<Ctrl+G>`
Interactively replace a text string	`<Alt+%>`
Using regular expressions	`<Alt+X> query-replace-regexp`
Replace this one, go on to next	`<Space> or y`
Replace this one, don't move	`,`
Skip to next without replacing	`<Del> or n`
Replace all remaining matches	`!`
Back up to the previous match	`^`
Exit query-replace	`<Enter>`
Enter recursive edit (`<Ctrl+Alt+C>` to exit)	`<Ctrl+R>`



```
UNIX is the operating system of choice for students.
UNIX is the operating system of choice for computer scientists.
UNIX is the operating system of choice for engineers.
UNIX is the operating system of choice for system administrators.
UNIX is the operating system of choice for web servers.
UNIX is the operating system of choice for scientists.
UNIX is the operating system of choice for networking.
UNIX is the operating system of choice for mathematicians.

U:---  osfile        All L8     (Fundamental)
Wrote /usr/home/bob/osfile
```

FIGURE 3.9 Display after Step 21.

3.3.6.1 Practice Session 3.10

Step 1: At the shell prompt, type **`emacs osfile`** and then press **`<Enter>`**.

Step 2: Type **`Windows is the operating system of choice for everyone.`**

Step 3: Press **`<Ctrl+@>`**. The mark is now set at the end of the line you typed in Step 2. Highlight the whole first line with the graphics cursor and left mouse button. This will define the region that will be put in the Kill Ring.

Step 4: Press **`<Esc+W>`**. This action copies the region to the Kill Ring.

Step 5: Press **`<Enter>`** to start a new line in the buffer, which should be blank. The cursor should be positioned at the start of this new line.

Step 6: Press **`<Ctrl+Y>`**. The first line of text is now pasted into the next blank line from the Kill Ring.

Step 7: Repeat Steps 5 and 6 six more times so that you now have eight lines of text in the buffer, all containing the text `Windows is the operating system of choice for everyone`.

Step 8: Position the cursor on the `W` in `Windows` on the first line of the buffer.

Step 9: Save the buffer at this point with **`<Ctrl+X> <Ctrl+S>`**.

Step 10: Press **`<Esc+X>`**. Then type **`query-replace`** and press **`<Enter>`**. These actions begin an interactive search and replace. The prompt `Query replace:` appears.

Step 11: Type **`everyone`** and then press **`<Enter>`**. The prompt `with:` appears.

Step 12: Type **`students`** and then press **`<Enter>`**. The prompt `Query replacing everyone with students: (? for help)` appears.

Step 13: Pressing **`<Space>`** on the keyboard replaces the word everyone on the first line with the word students, and the prompt `Query replacing everyone with students: (? for help)` appears again.

Step 14: Press **`<Enter>`**. The prompt `Replaced 1 occurrence` appears.

Step 15: Position the cursor over the e in the word everyone on the second line of the buffer.

Step 16: Repeat Steps 10–14, interactively replacing the word `everyone` each time it appears with the words computer scientists, engineers, system administrators, `web servers`, `scientists`, `networking`, and `mathematicians` on lines 2–8 of the buffer.

Be sure that the second through eighth times you do Step 16, you always position the cursor on the previous line to the current line you want to replace text on!

Step 17: Position the cursor on the `W` in `Windows` on the first line of the buffer.

Step 18: Press **`<Esc+X>`**. Then type **`replace-string`** and press **`<Enter>`**. These actions begin an unconditional search and replace. The prompt `replace string:` appears.

Step 19: Type **`Windows`** and then press **`<Enter>`**. The prompt `Replace string Windows with:` appears.

Step 20: Type UNIX and then press **`<Enter>`**. The prompt `Replaced 8 occurrences` appears. Correct?

Step 21: Save the buffer with **`<Ctrl+X><Ctrl+S>`**, print it using the facilities available on your computer system, and exit emacs with **`<Ctrl+X> <Ctrl+C>`**. Your screen display should appear like Figure 3.9.

The following in-chapter exercises ask you to apply some of the operations you learned about in the previous practice sessions:

EXERCISE 3.8

Run emacs and define keyboard macro commands that automatically delete

a. every other word in a line of unspecified length,

b. every other line in a file of unspecified length,

c. every other word and every other line in a file of unspecified length with lines of unspecified length.

EXERCISE 3.9

Write a keyboard macro, as shown in Section 3.3.5, to do everything shown in Steps 10–14 of Practice Session 3.10.

To get some further practice with emacs, do Problem 16 at the end of the chapter.

3.3.7 How to Do Purely Graphical Editing with GNU Emacs

Up to this point in our work, it was possible to use emacs in a single text-based terminal window and obtain the results shown. If you connect to UNIX by using one of the methods described in Chapter 2 you are likely interfacing with the operating system via an intermediary known as the X Window System, which would allow you to do all of your emacs work in a graphical environment. We present more information on the X Window System and some of its facilities (e.g., the particular features of the GUI) in Chapter 22. For the purposes of learning emacs—if you are using UNIX and the X Window System—you may be able to run emacs in its own frame on your screen display or possibly in several frames on your screen display simultaneously. Be aware that in our base UNIX systems, PC-BSD and Solaris, the latest version of GNU graphical emacs version is available through the package management facilities in those systems, as described in Chapter 23.

3.3.8 Editing Data Files

The following practice session demonstrates the use of graphical GNU emacs to do some further editing of the datafile created in Practice Session 3.9. The look and feel of GNU emacs, running under the X Window System using the KDE or Gnome desktop default windowing environments in PC-BSD and Solaris, is very similar to a word processor or desktop publishing application running under any other operating system that has a GUI, such as Windows 10 or Mac OS X. In the practice sessions that follow, we are using GNU emacs 24.3.1.

3.3.8.1 Practice Session 3.11

Step 1: In a terminal or console window, at the shell prompt, type **`emacs datafile`** and then press **`<Enter>`**. Your screen display should look similar to the one shown in Figure 3.8.

Step 2: Use the mouse to position the cursor over the character `1` at the beginning of the tenth line in the buffer, and then click the left mouse button. The cursor is now positioned over the character `1`.

Step 3: Click and hold down the left mouse button over the character `1`, and then drag the mouse so that the entire tenth line is highlighted, including one character to the right of the `0` in the number `10` at the end of the line. Release the left mouse button.

The whole first line should be highlighted.

Step 4: Position the cursor with the mouse so that the arrow points to the menu choice **Edit** at the top of the emacs screen. Click the left mouse button. A set of pull-down menu choices appears, similar to that shown in Figure 3.10.

Step 5: Make the **Copy** menu choice. The text that you highlighted (selected) in Step 3 is now held in a temporary buffer.

Step 6: Press **`<Enter>`**. This opens an eleventh line at the bottom of the buffer.

Step 7: Move the mouse so that the cursor is over the first character position on the eleventh line, and click the left mouse button. The cursor is now in that position in the buffer.

Step 8: Make the pull-down menu choice **Edit>Paste**. You have now pasted the 10 characters from the tenth line in the buffer into the eleventh line in the buffer. Your screen display should now look similar to Figure 3.11.

Step 9: Make the pull-down menu choice **File>Save (current buffer)**. In the `Write file:` dialog box that appears, save the file in your home directory as **datafile11**, and then make the pull-down menu choice **File>Exit Emacs**.

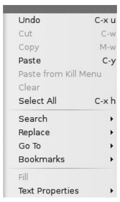
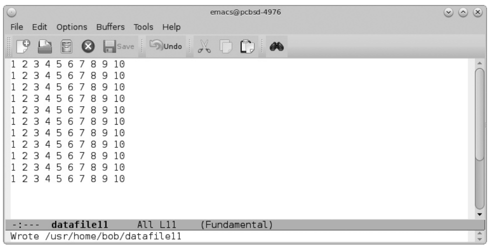


FIGURE 3.10 The emacs Edit pull-down menu.



```
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10

-:---  datafile11      All L11     (Fundamental)
Wrote /usr/home/bob/datafile11
```

FIGURE 3.11 Datafile after editing and adding an 11th line.

3.3.9 How to Start, Save a File, and Exit in Graphical Emacs

As illustrated in Practice Session 3.11, this text editor gives you exclusive mouse/GUI expediency. This method of working on a text file is most efficient for beginners as well as experienced users. In doing Practice Session 3.12, which starts by editing the file you created in Practice Session 3.7, you will be able to compare the speed and ease of operations using keystroke commands to those of mouse/GUI interaction. Note that, on the pull-down menu shown in Figure 3.10, keystroke commands also are shown for some of the menu choices. Clicking the menu choice button or pressing the keyboard keys would accomplish the same thing. This flexibility adds to the ease of your use of emacs.

Practice Session 3.12 for PC-BSD lets you edit the file **alien** that you created at the start of this section in Practice Session 3.7. That practice session allowed you to use emacs to create a simple C-shell script file of aliases. You will now modify it so that it can be used as aliases for the Bourne shell. You will also modify the existing file **.shrc** in your home directory so that when you log in and are using the Bourne shell, you have the aliased commands in the file **alien3** available to you. Before you begin Practice Session 3.12, take the following preparatory steps (if you are using Solaris, modify the **.bashrc** file in your home directory and proceed through this practice session for Bash rather than the Bourne shell):

Preparatory Step 1: Use the **`ls -la`** command to find out if you have a **.shrc** file in your home directory. If you have no **.shrc** file in your home directory, then use emacs to create a new file named **.shrc** with nothing in it. Then exit emacs, and type **`chmod u+X .shrc`** and press **`<Enter>`**.

Preparatory Step 2: Find out which shell you are currently using by typing **`echo $SHELL`** and pressing **`<Enter>`**. If you are using the C shell, the system will respond with `/bin/csh`. If you are using the Bourne shell, the system will respond `/bin/sh`.

Preparatory Step 3: If you are <u>not</u> using the Bourne shell as determined in Step 2, switch to the Bourne shell at the existing shell prompt by typing **`sh`** and then pressing **`<Enter>`**.

3.3.9.1 Practice Session 3.12

Step 1: At the shell prompt, type **`emacs alien`** and then press **`<Enter>`**. The file that you created in Practice Exercise 3.7 is loaded into the buffer, and the contents of the emacs buffer looks like the one shown in Figure 3.6. Use the cursor and mouse for cursor positioning and the keyboard keys for text entry and deleting to modify the file so that it looks like this:

```
#DOS aliases

alias del="rm"

alias dir="ls -la"

alias type="more"
```

Step 2: Position the cursor, using the mouse and left mouse button, to the right of the double-quote character (**"**) at the end of the third line.

Step 3: Press **`<Enter>`** to open a blank line and put the cursor at the beginning of the line.

Step 4: Type **`alice dirw="ls"`**.

Step 5: Position the cursor, using the mouse and left mouse button, at character `a` in `alice`.

Step 6: Hold down the left mouse button and move the mouse so that the word `alice` and the following space are highlighted. At the top of the screen, make the **Edit** pull-down menu choice **Cut** to cut the word `alice` from the buffer.

Step 7: Type **`alias`**.

Step 8: Move the mouse so that the cursor is over the second a character in the word `alias`. Click the left mouse button.

Step 9: Press the **`<Down>`** arrow key on the keyboard twice. The cursor should now be at the beginning of the blank line below the line that reads **`alias type="more"`**.

Step 10: From the **Edit** pull-down menu, choose **Paste**. The cut word `alice` has been put back into the buffer at the start of the line.

Step 11: Use the mouse and left mouse button to position the cursor at the end of the word `alice`, after the character `e`.

Step 12: Use the **`<Delete>`** or **`<Backspace>`** keys to delete the letters `c` and `e` from the word `alice`.

Step 13: Type **`as copy="cp"`**.

Step 14: Continue moving the cursor to the proper positions and add the necessary characters.

Step 15: From the pull-down menu File, choose **Save Buffer As**…

Step 16: In the `Write file:` dialog box that opens on screen, save the file as **alien3**.

Step 17: From the File pull-down menu, make the choice **File**>**Open File**. In the `Find file:` dialog box that opens, put a check mark in the box that is for `Show Hidden Files`.

Locate and select the **.shrc** file (which should be in your home directory) and make the **Open** choice. A new buffer opens on screen containing the contents of the **.shrc** file. Position the cursor anywhere on a blank line in the buffer for the file **.shrc**.

Step 18: From the File pull-down menu, make the choice **File**>**Insert file**… In the dialog box that opens, choose **alien3** and insert it. The lines of text from **alien3**'s DOS aliases should now be inserted into the file **.shrc** at the location you designated in Step 17.

Step 19: From the pull-down menu File, make the choice **File>Save (current buffer)**.

Step 20: Make the pull-down menu choice **File>Exit Emacs** to quit emacs and return to the shell prompt.

Step 21: Test your new **.shrc** file in a terminal or console window. For PC-BSD, at the Bourne shell prompt, type **`. $HOME/.shrc`** and press **`<Enter>`**. Then, test all of the aliased commands and note the results. For example, if you type **`dir`**, you should get the results of the **`ls -la`** command that is executed in the current working directory. For Solaris, close and then reopen the terminal window to reinitiate the shell, and test the new aliases.

Since for PC-BSD the Bourne shell run in the practice session is an *interactive shell*, not your login shell, in order to make the commands that you placed in your ~/**.shrc** file work, you must execute the Step 21 procedure of typing `. $HOME/.shrc` and pressing `<Enter>`!

In Problem 14 at the end of the chapter, we ask you to do a similar set of operations done in Practice Session 3.12 for Bash, but exclusively for users of PC-BSD.

3.3.10 Emacs Graphical Menus

Figures 3.12 and 3.13 show the contents of another two of the most important pull-down menus in a graphical emacs: Files and Tools. To the right of each pull-down choice is the keystroke command equivalent, if there is one.

To get some further practice with a graphical emacs, do Problem 18 at the end of this chapter.

3.3.11 Creating and Editing C Programs

Besides being a powerful text editor/word processor, emacs can do multiple chores that are useful to a computer user from within the emacs program itself, such as composing e-mail, executing shell scripts, Internet work, and program development in C, HTML, and Java. Since the text for anything more than a trivial program must be generated in a text editor

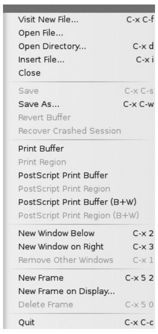


FIGURE 3.12 The Files pull-down menu.

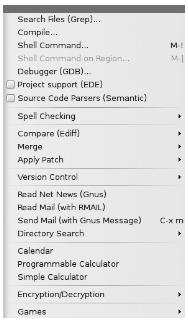


FIGURE 3.13 The Tools pull-down menu.

of some sort, it stands to reason that this editor should also be able to compile, link, debug, and keep a record of source code revisions, as well as execute the program itself. This is easily done in emacs using some of its built-in capabilities. These kinds of all-in-one capabilities are present because in the days of character-only terminals and consoles, instead of leaving the editor to accomplish a chore outside of it, you could accomplish common tasks from within the editor. In modern UNIX, we can now simply switch between windows and never leave the editor. But it is still very useful to be able to harness some of the multiple capabilities of the program, mainly for the sake of efficiency.

Practice Session 3.13 allows you to type in the source code of a C program and use the special facilities of the editor to properly indent the text, compile and link the source code, and implement revisions according to compile-time errors. You can then execute the program in a terminal window to test it. The purpose of the program is to allow the user to type in an integer, and then another integer, and the first integer will be raised to the power indicated by the second integer.

A note about paths: If the path for the shell you are executing includes the working directory where the compiled and linked executable program is saved by emacs, then you can run the program. Otherwise, you will have to include the path to this directory. For example, emacs saved the source code and executable files in the current working directory `/root` on our system when we did Practice Session 3.13. Before running the program, since we knew we were running under the C shell, we checked the path variable by typing `echo $PATH`. The path display included the current working directory where emacs was saving our files. See Chapter 2, Sections 2.7 and 2.8, for information on how to view the path and set the path variable. Also see Chapter 17 for more about UNIX tools for the software development process. The source code for the program is as follows.

```
#include <math.h>
main()
{
     float x,y;
     printf("This program takes x and y values from stdin and
displaysx^y.\n");
     printf("Enter integer x: ");
```

```
        scanf("%f", &x);
        printf("Enter integer y: ");
        scanf("%f", &y);
        printf("x^y is: %6.3f\n", pow((double)x,(double)y));
  }
```

3.3.11.1 Practice Session 3.13

Step 1: At the shell prompt, type **`emacs power.c.`** Notice that the major mode for this new buffer is set to C/l mode.

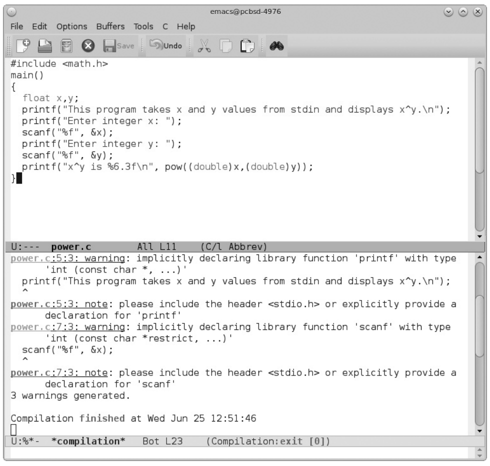
Step 2: Type in the program source code exactly as shown. Use the **`<Tab>`** key to produce the indentation shown in the C source code. Your emacs screen display should look similar to Figure 3.14.

Step 3: From the pull-down menus, make the choice **File**>**Save (current buffer)**.

Step 4: From the pull-down menus, make the choice **Tools**>**Compile**… In the minibuffer, the prompt `Compile command: make -k` appears. Use the backspace key to erase `make -k`, and then, to replace it, type **`cc power.c -lm -o power`**. A new buffer window appears in the emacs frame, showing the progress of the compilation/linking process.

Step 5: From the pull-down menus, make the choice **Tools**>**Compile**… In the minibuffer, the prompt `Compile command: cc power.c -lm -o power` should appear. Press **`<Enter>`** to accept this compile/link command.

If you made mistakes in typing the C code, repeat Steps 2 through 5 <u>until you get no error messages that prevent compilation and linkage!</u> The bottom buffer window of



```
#include <math.h>
main()
{
  float x,y;
  printf("This program takes x and y values from stdin and displays x^y.\n");
  printf("Enter integer x: ");
  scanf("%f", &x);
  printf("Enter integer y: ");
  scanf("%f", &y);
  printf("x^y is %6.3f\n", pow((double)x,(double)y));
}

U:---  power.c        All L11    (C/l Abbrev)
power.c:5:3: warning: implicitly declaring library function 'printf' with type
      'int (const char *, ...)'
  printf("This program takes x and y values from stdin and displays x^y.\n");
  ^
power.c:5:3: note: please include the header <stdio.h> or explicitly provide a
      declaration for 'printf'
power.c:7:3: warning: implicitly declaring library function 'scanf' with type
      'int (const char *restrict, ...)'
  scanf("%f", &x);
  ^
power.c:7:3: note: please include the header <stdio.h> or explicitly provide a
      declaration for 'scanf'
3 warnings generated.

Compilation finished at Wed Jun 25 12:51:46

U:%*-  *compilation*   Bot L23    (Compilation:exit [0])
```

FIGURE 3.14 Display after Step 2.

Figure 3.14 shows *warning messages*, but not exceptions that prevented compilation and linkage.

Step 6: If all syntax errors have been removed from the `power.c` source code, you should get a screen display similar to Figure 3.14, which indicates in the bottom buffer window that you have successfully compiled and linked `power.c`.

Step 7: You can now exit emacs, and in a terminal window test the program by typing **power** on the command line. Remember that the path must be set for the current shell so that executable programs in the directory the file power is in will run.

3.3.12 Working in Multiple Buffers

As seen in previous exercises, it is possible to insert one buffer into another and to open windows into different buffers, some of which may not even contain text you want to edit, at the same time. This capability is important when you want to compose the contents of a buffer or file from perhaps many other buffers or files that you have previously created. The following practice session shows you how to create, move between, and copy and paste between several buffers open within one emacs frame.

3.3.12.1 Practice Session 3.14

Step 1: Create a subdirectory under your home directory named **multi**, and make that subdirectory the current working directory.

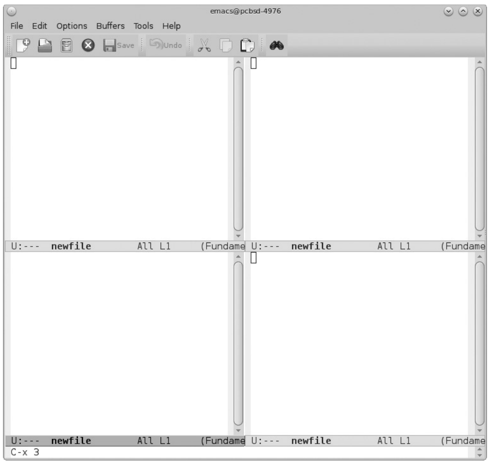
Step 2: At the shell prompt, type **`emacs newfile`**. You should now be editing the buffer **newfile** with a single window.

Step 3: In emacs, make the pull-down menu choice **File>New Window Below**. The frame should now be split horizontally, so that you have two windows, one above the other, both showing the contents of **newfile**.

Step 4: Click with the mouse in the upper window, and then press **`<Ctrl+X>` 3**. The upper window from Step 3 should now be split vertically into two windows, showing you a total of three windows into the buffer **newfile**.

Step 5: Repeat Step 4 in the lower window of the frame. You should now have four windows showing the contents of the buffer **newfile**. Your screen display should look similar to Figure 3.15. If you did Steps 1–4 incorrectly, you can always use the **File>Remove Other Windows** pull-down menu choice to return you to a single window display, and then try again.

Step 6: Click the mouse in the upper-left window and type **1 2 3 4 5**. Then make the pull-down menu choice **File>Save As**. In the `Write file:` dialog box that appears on screen, type **`firstrow`** in the `Name:` box, and then use the `Name` dialog pane and double left-click on the folder **multi**. Then, make the choice OK. A new file named **firstrow** is created on disk in the directory named **multi**, and you are still seeing four windows into that buffer.

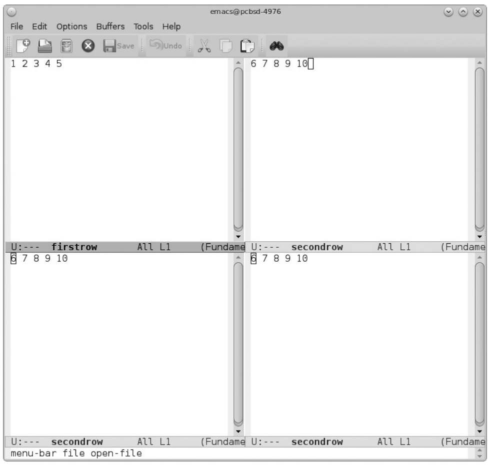


```
U:---  newfile        All L1     (Fundame U:---  newfile        All L1     (Fundame
U:---  newfile        All L1     (Fundame U:---  newfile        All L1     (Fundame
C-x 3
```

FIGURE 3.15 Display after Step 5.

Step 7: Click the mouse in the upper-right window, position the cursor at the right after the 5, and use the **<Backspace>** or **<Delete>** keys to erase the numbers 1, 2, 3, 4, and 5. Then type **6 7 8 9 10**. Then make the pull-down menu choice **File>Save As**. In the Write file: dialog box that appears on screen, type **secondrow** in the Name: box. The file will be saved in the folder **multi**. Then make the choice **OK**.

Step 8: Click the mouse in the upper-left window. Make the pull-down menu choice **File>Open File**. In the Find file: dialog box that appears on screen, highlight the file **firstrow** in the Name dialog pane. Then make the choice **Open**. You now should have a screen display similar to Figure 3.16, with the upper-left window showing the contents of **firstrow**, and the remaining three windows showing the contents of **secondrow**.



```
1 2 3 4 5                                 6 7 8 9 10
U:---  firstrow       All L1     (Fundame U:---  secondrow      All L1     (Fundame
6 7 8 9 10                                6 7 8 9 10
U:---  secondrow      All L1     (Fundame U:---  secondrow      All L1     (Fundame
menu-bar file open-file
```

FIGURE 3.16 Display after Step 8.

Step 9: Click the mouse in the lower-left window, position the cursor to the right of the `0`, erase `6`, `7`, `8`, `9`, and `10`, and type **11 12 13 14 15**. Then make the pull-down menu choice **File>Save As**. In the `Write file:` dialog box that appears on screen, type **`thirdrow`** in the `Name:` box. Then make the choice **`OK`**.

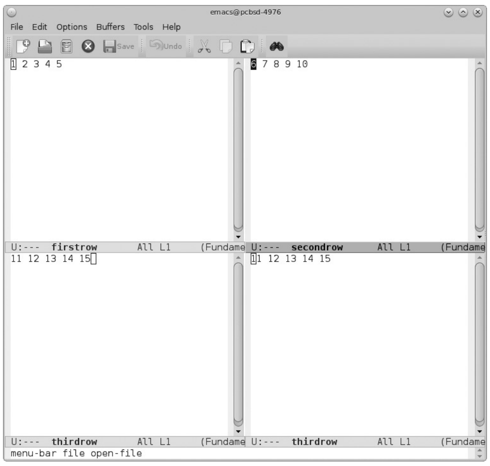
Step 10: Click in the upper-right window and make the pull-down menu choice **File>Open File**. In the `Find file:` dialog box that appears on screen, highlight the file **secondrow** in the `Name` dialog pane. Then, make the choice **Open**.

Step 11: Click in the lower-left window, and make the pull-down menu choice **File>Open File**. In the minibuffer, type **`thirdrow`**. Your screen display should now look similar to Figure 3.17.

Step 12: Click the mouse in the lower-right window, and make the pull-down menu choice **File>Save As**. In the `Write file:` dialog box, type **`four`** in the `Name:` box. Make the choice **OK**. Click in the lower-left window, and make the pull-down menu choice **File>Open File**. In the `Find file:` dialog box that opens, highlight **thirdrow**, and open it.

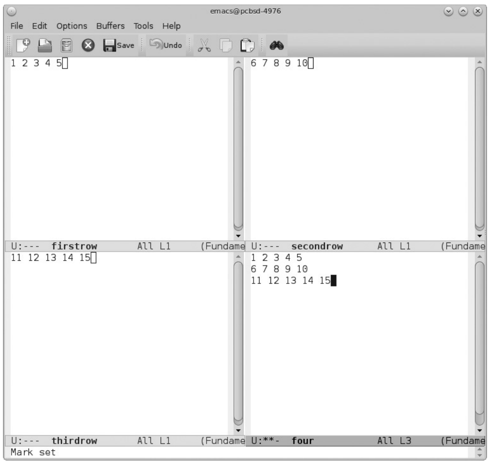
Step 13: Click the mouse in the lower-right window, and use the backspace key to erase `11`, `12`, `13`, `14`, and `15`. Then use the pull-down menu choices **Edit>Copy** and **Edit>Paste** to copy `1 2 3 4 5`, `6 7 8 9 10`, and `11 12 13 14 15` onto the first three rows of the lower-right window. Your screen display should look similar to Figure 3.18.

Step 14: Finally, with the lower-right window the current window, make the pull-down menu choice **File>Save As**. In the `Write file:` dialog box, in the `Name` box, type **`four`**. Overwrite the old buffer **four**. Then quit emacs without saving any of the buffers.



```
1 2 3 4 5                                  6 7 8 9 10

U:---  firstrow      All L1     (Fundame   U:---  secondrow     All L1     (Fundame
11 12 13 14 15                             11 12 13 14 15

U:---  thirdrow      All L1     (Fundame   U:---  thirdrow      All L1     (Fundame
menu-bar file open-file
```

FIGURE 3.17 Display after Step 11.



```
1 2 3 4 5                                   6 7 8 9 10
U:---  firstrow      All L1     (Fundame    U:---  secondrow     All L1     (Fundame
11 12 13 14 15                              1 2 3 4 5
                                            6 7 8 9 10
                                            11 12 13 14 15
U:---  thirdrow      All L1     (Fundame    U:**-  four          All L3     (Fundame
Mark set
```

FIGURE 3.18 Display after Step 13.

3.3.13 Changing Emacs Behavior

This section describes the basic methods of customizing and modifying the behavior of GNU emacs. This includes the following operations:

- Using the Options menu to modify options.
- Using Custom (a GUI-based interface) to change preferences and options, and in conjunction with that interface, also using the traditional typed `<Alt+X> customize` command set.
- Writing keystroke abbreviations with `abbrev`.
- Writing keystroke macro commands, as shown in Section 3.3.5.
- Redefining keyboard keys.
- Writing emacs Lisp (elisp) code to customize the behavior of emacs, and entering that code directly into your **~/.emacs** startup configuration file.

All of these operations can make changes to your **~/.emacs** startup configuration file to give you a more customized and personalized emacs session, one tailored to your particular needs and methods of entering text for a particular application. Also, as will be seen, elisp code is generated by what operations you do. But you don't really need to know any of the details of how to program in elisp to actually achieve all of the these operations!

The following subsections describe and give examples of all of the given operations. In addition, Tables 3.15 and 3.16 give a summary of the important keystrokes that implement `<Alt+X> customize`, keystroke abbreviations with `abbrev`, and writing keystroke macros.

TABLE 3.15 Some Ways to Change Emacs Behavior

Customization Action	Keystrokes
Abbrevs	
Add global abbrev	`<Ctrl+X> a g`
Add mode-local abbrev	`<Ctrl+X> a l`
Add global expansion for this abbrev	`<Ctrl+X> a i g`
Add mode-local expansion for this abbrev	`<Ctrl+X> a i l`
Explicitly expand abbrev	`<Ctrl+X> a e`
Expand previous word dynamically	`<Alt+/>`
Macros	
Start defining a keyboard macro	`<Ctrl+X> ( or <F3>`
End keyboard macro definition	`<Ctrl+X> ) or <F4>`
Execute last-defined keyboard macro	`<Ctrl+X> e or <F4>`
Append to last keyboard macro	`<Ctrl+U> <Ctrl+X> (`
Name last keyboard macro	`<Alt+X> name-last-kbd-macro`
Insert Lisp definition in buffer	`<Alt+X> insert-kbd-macro`
Customize variables and faces	`<Alt+X> customize`
Simple customization with `<Alt+X>` customize	
`(global-set-key (kbd "<Ctrl+C> g") 'search-forward)`	
`(global-set-key (kbd "<Alt+#>") 'query-replace-regexp)`	

3.3.13.1 Using the Options Menu

The easiest and quickest way to customize the behavior of emacs is by using the GNU emacs pull-down menu choices under Options, which is shown in Figure 3.19. For example, if you put a check mark next to the **Highlight Matching Parentheses** choice, all matching left and right parentheses in the buffer will be highlighted as you type them.

This option will only be true for the current session of emacs. If you want to retain this option for all future sessions of emacs, make the Options menu choice **Save Options**. The following valid line of elisp will automatically be written to your **~/.emacs** file, under the `custom-set-variables` group.

```
`(show-paren-mode t)
```

EXERCISE 3.10

Show the emacs Help facility keystroke sequence you would use to find out what the option `show-paren-mode` is. Then, list the first few lines of how the Help facility describes the `show-paren-mode` option.

You can also customize by group from the **Options** menu, if you make the **Customize Emacs** choice, and then make any of the subchoices below that. For example, if you make the **Options>Customize Emacs>Top-Level Customization Group** choice, a new buffer opens on screen, and allows you to select from all of the subgroups of custom variables.

The next section shows how to achieve this kind of customization as a typed command.

TABLE 3.16 Keystroke Macros

Keystrokes	Command Name	Action
`<Ctrl+X> (`	`kmacro-startmacro`	Start macro definition.
`<F3>`	`kmacro-startmacro-or-insertcounter`	Start macro definition. If pressed while defining a macro, insert a counter.
`<Ctrl+X> )`	`kmacro-end-macro`	End macro definition.
`<F4>`	`kmacro-end-orcall-macro`	End macro definition (if definition is in progress) or invoke last keyboard macro.
`<Ctrl+X> e`	`kmacro-end-andcall-macro`	Execute last keyboard macro defined. Can type `e` to repeat macro.
`<Ctrl+X> <Ctrl+K> n`	`name-last-kbdmacro`	Name the last macro you created (before saving it).
(none)	`insert-kbd-macro`	Insert the macro you named into a file.
(none)	`macroname`	Execute a named keyboard macro.
`<Ctrl+X> q`	`kbd-macro-query`	Insert a query in a macro definition.
`<Ctrl+u> <Ctrl+X> q`	(none)	Insert a recursive edit in a macro definition.
`<Ctrl+Alt+C>`	`exit-recursive-edit`	Exit a recursive edit.
`<Ctrl+X> <Ctrl+K> b`	`kmacro-bind-tokey`	Bind a macro to a key (`<Ctrl+X> <Ctrl+K>` 0-9 and `A-Z` are reserved for macro bindings). Lasts for current session only.
`<Ctrl+X> <Ctrl+K> Space`	`kmacro-step-editmacro`	Edit a macro while stepping through it.
`<Ctrl+X> <Ctrl+K> l`	`kmacro-editlossage`	Turn the last 100 keystrokes into a keyboard macro.
`<Ctrl+X> <Ctrl+K> e`	`edit-kbd-macro`	Edit a keyboard macro by typing `<Ctrl+X> e` for the last keyboard macro defined, `<Alt+X>` for a named macro, `<Ctrl+H>` l for lossage, or keystrokes for a macro bound to a key.
`<Ctrl+X> <Ctrl+K> Enter`	`kmacro-editmacro`	Edit the last keyboard macro.
`<Ctrl+X> <Ctrl+K> <Ctrl+E>`	`kmacro-editmacro-repeat`	Edit the last keyboard macro again.
`<Ctrl+X> <Ctrl+K> <Ctrl+T>`	`kmacro-swap-ring`	Transpose last keyboard macro with previous keyboard macro.
`<Ctrl+X> <Ctrl+K> <Ctrl+D>`	`kmacro-deletering-head`	Delete last keyboard macro from the macro ring.
`<Ctrl+X> <Ctrl+K> <Ctrl+P>`	`kmacro-cycle-ringprevious`	Move to the previous macro in the macro ring.
`<Ctrl+X> <Ctrl+K> <Ctrl+N>`	`kmacro-cycle-ringnext`	Move to the next macro in the macro ring.
`<Ctrl+X> <Ctrl+K> <Ctrl+R>`	`apply-macro-toregion-lines`	Apply this macro to each line in a region.

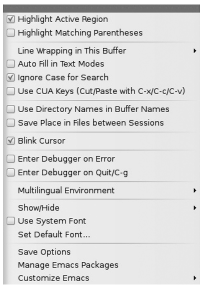
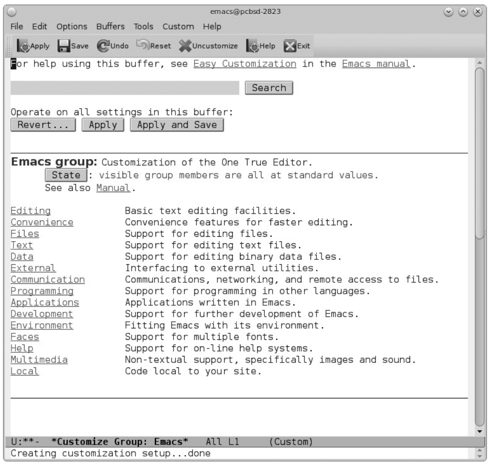

☑ Highlight Active Region
☐ Highlight Matching Parentheses
Line Wrapping in This Buffer ▸
☐ Auto Fill in Text Modes
☑ Ignore Case for Search
☐ Use CUA Keys (Cut/Paste with C-x/C-c/C-v)
☐ Use Directory Names in Buffer Names
☐ Save Place in Files between Sessions
☑ Blink Cursor
☐ Enter Debugger on Error
☐ Enter Debugger on Quit/C-g
Multilingual Environment ▸
Show/Hide ▸
☐ Use System Font
Set Default Font...
Save Options
Manage Emacs Packages
Customize Emacs ▸

FIGURE 3.19 Options menu choices.


emacs@pcbsd-2823
File Edit Options Buffers Tools Custom Help
Apply Save Undo Reset Uncustomize Help Exit

```
For help using this buffer, see Easy Customization in the Emacs manual.

                                                 Search

Operate on all settings in this buffer:
 Revert...   Apply   Apply and Save

Emacs group: Customization of the One True Editor.
      State : visible group members are all at standard values.
      See also Manual.

Editing             Basic text editing facilities.
Convenience         Convenience features for faster editing.
Files               Support for editing files.
Text                Support for editing text files.
Data                Support for editing binary data files.
External            Interfacing to external utilities.
Communication       Communications, networking, and remote access to files.
Programming         Support for programming in other languages.
Applications        Applications written in Emacs.
Development         Support for further development of Emacs.
Environment         Fitting Emacs with its environment.
Faces               Support for multiple fonts.
Help                Support for on-line help systems.
Multimedia          Non-textual support, specifically images and sound.
Local               Code local to your site.

U:**-  *Customize Group: Emacs*   All L1     (Custom)
Creating customization setup...done
```

FIGURE 3.20 Emacs customization groups.

3.3.13.2 Changing emacs Variables with Custom and the `<Alt+X> customize` *Command*

Emacs has many settings that you can change. Most settings are customizable via affecting the settings of *variables*, which are also called *user options*. There is a huge number of customizable variables, controlling numerous aspects of emacs behavior. A separate class of settings, which we do not cover here, are the *faces*, which determine the fonts, colors, and other attributes of text.

To browse and alter settings (both variables and faces), at the emacs command prompt type `<Alt+X> customize`. This creates a customization buffer, which lets you navigate through a logically organized list of settings, edit and set their values, and save them permanently.

Customization settings are organized into *customization groups*. These groups are collected into bigger groups, all the way up to a master group called Emacs, shown near the top of the buffer in Figure 3.20.

`<Alt+X> customize` creates a *customization buffer* that looks similar to Figure 3.20.

If you are interested in customizing a particular setting or customization group that you already know the name of, you can go straight there with the commands `<Alt+X> customize-option`, `<Alt+X> customize-face`, or `<Alt+X> customize-group`.

The main part of the buffer in Figure 3.20 shows the "Emacs" customization group, which contains several other subgroups ("Editing," "Convenience," etc.). The contents of those subgroups are shown in the single line of description for each group.

The *state* of the group indicates whether the settings in that group have been edited, set or saved.

Most of the customization buffer cannot be changed, but it includes many editable fields. For example, at the top of the customization buffer is an editable field for searching for settings, with a `Search` button next to it. There are also buttons and links that you can activate by either clicking with the mouse, or moving the point there and then pressing `<Enter>`. For example, group names like "[Editing]" are links; activating one of these links brings up another customization buffer for that group.

In any particular customization buffer, you can type `<Tab>` (widget-forward) to move forward to the next button or editable field. `<Shift+Tab>` (widget-backward) moves back to the previous button or editable field.

3.3.13.2.1 Browsing and Searching for Settings: From the top-level customization buffer created by `<Alt+X> customize`, you can follow the links to the subgroups of the "Emacs" customization group. These subgroups may contain settings for you to customize; they may also contain further subgroups, dealing with yet more specialized subsystems of emacs. As you graphically navigate the hierarchy of customization groups, you will find some settings that you want to customize according to your own personal preferences, and according to the nature of the text documents that you are efficiently trying to edit.

3.3.13.2.2 Changing a Variable: Here is an example of what a variable, or user option, looks like in a specific customization buffer. This variable is accessed by descending down from the top emacs group through the groups `Editing>Killing`, and then left-clicking on the small diamond shape pointing toward the text `Kill Ring Max`:

```
Kill Ring Max: 60
[State]: STANDARD.
Maximum length of kill ring before oldest elements are thrown
away.
```

The first line shows that the variable is named `kill-ring-max`, formatted as `Kill Ring Max` for easier viewing. Its value is 60. On our graphical display, the line after the variable name indicates the customization state of the variable: in this example, `STANDARD` means you have not changed the variable, so its value is the default one. The `[State]` button gives a menu of operations for customizing the variable.

Below the customization state is the documentation for the variable. To enter a new value for `Kill Ring Max`, just click to the right of the value and edit it. As you begin to alter the text, the `[State]` line will change to:

```
[State]: EDITED, shown value does not take effect until you
set or save it.
```

Editing the value does not make it take effect right away. To do that, you must set the variable by left-clicking on the `[State]` button and choosing **Set for Current Session**. Then the variable's state becomes:

```
[State]: SET for current session only.
```

At this point, you could have made the menu choice **Save for Future Sessions**.

Also, you don't have to worry about specifying a value that is not valid; the **Set for Current Session** operation checks for validity and will not install an unacceptable value.

When you set a variable, the new value takes effect only in the current emacs session. To save the value for future sessions, use the `[State]` button and select the **Save for Future Sessions** operation. Saving custom settings works by writing elisp code to a file, most often your **~/.emacs** file. Future emacs sessions automatically read this file at startup, which invokes and establishes the customizations again.

You can also restore the variable to its standard value by using the `[State]` button and selecting the **Erase Customization** menu choice. There are four reset operations as follows.

- Undo edits: If you have modified but not yet set the variable, this restores the text in the customization buffer to match the actual value.
- Reset to saved: This restores the value of the variable to the last saved value, and updates the text accordingly.
- Erase customization: This sets the variable to its standard value. Any saved value that you have is also erased.
- Set to backup value: This sets the variable to a previous value that was set in the customization buffer in this session. If you customize a variable and then reset it, which discards the customized value, you can get the discarded value back again with this operation.

Sometimes it is useful to record a comment about a specific customization. Use the `Add Comment` item from the `[State]` menu to create a field for entering the comment.

3.3.13.2.3 Globally Saving Customizations for a Group: Near the top of any group's customization buffer, you can save all customization settings shown in that group buffer by choosing either the `[Apply]` or `[Apply and Save]` buttons. `[Apply]` only saves for the current session, and `[Apply and Save]` saves for future sessions by modifying the **~/.emacs** file accordingly by putting elisp code in the **~/.emacs** file.

3.3.13.2.4 More about Emacs Variables: A *variable* is an elisp symbol that has a value. The symbol's name is the *variable name*. A variable name can contain any characters that can appear in a file, but most variable names consist of ordinary words separated by hyphens.

The name of the variable is descriptive of its role in the emacs environment. Most variables also have a documentation string, which describes what the variable's purpose is, what kind of value it should have, and how the value will be used.

You can view the documentation for a variable, such as **`somevariablename`**, using the help command **`<Ctrl+H> v`** *`Describe variable:`* **`somevariablename`** in the minibuffer. To use this facility, type in the command **`<Ctrl+H> v`**; the system prompts you in the minibuffer with *`Describe variable:`*; then type in the variable name, such as **`somevariablename`**, and press **`<Enter>`**.

Elisp uses variables for internal record keeping, but as noted earlier, the most interesting variables for a user who will not be writing elisp programs per se are those meant for users to change—these are called *customizable variables* or *user options*.

Elisp allows any variable (with a few exceptions) to have any kind of value. However, many variables are meaningful only if assigned values of a certain type. For example, as shown in Section 3.3.13.2.2, only numbers are meaningful values for `kill-ring-max`, which specifies the maximum length of the kill ring; if you assign `kill-ring-max` a text string as a value, commands such as `<Ctrl+Y>` (yank) will signal an error. On the other hand, some variables don't care about what kind or type of value you assign them; for instance, if a variable has one effect for nil values and another effect for non-nil values, then any value that is not the symbol `nil` induces the second effect, regardless of its type (by convention, we usually use the value `t`—a symbol that stands for "true"—to specify a non-nil value). If you set a variable using the customization buffer, you need not worry about giving it an invalid type: the customization buffer usually only allows you to enter meaningful values. When in doubt, use **`<Ctrl+H> v`** *`Describe variable:`* **`somevariablename`** to check the variable's documentation string to see the kind of value it expects.

3.3.13.2.5 Examining and Setting Variables: The following are some examples of how to examine and set the values of user options. The first general form of this syntax is:

```
<Ctrl+H> v variablename <Enter>
```

This general form uses emacs help function with the v option and displays the value and documentation for variable `variablename`.

The second general form achieves the change in the variables value:

```
<Alt+X> set-variable <ENTER> var <ENTER> value <ENTER>
```

This changes the value of variable `var` to `value`.

It reads a variable name that you supply by typing in the minibuffer, with completion, and displays both the value and the documentation of the variable. For example:

```
<Ctrl+H> v fill-column <ENTER>
```

A new buffer opens and displays the following:

```
        fill-column is a variable defined in 'C source code'.
        Its value is 70

            Automatically becomes buffer-local when set.
            This variable is safe as a file local variable if
its value
            satisfies the predicate 'integerp'.
        Documentation:

        Column beyond which automatic line-wrapping should
happen.
        Interactively, you can set the local value with <Ctrl+X>
f

        You can customize this variable.
```

Click on the underlined text `customize` and you can use a buffer to change the value of this variable.

The most convenient keystroke method to set a specific customizable variable is by typing `<Alt+X> set-variable`. This reads the variable name with the minibuffer (with completion), and then reads an elisp expression for the new value that you type in the minibuffer a second time (you can insert the old value into the minibuffer for editing via `<Alt+N>`). For example:

```
<Alt+X> set-variable <ENTER> fill-column <ENTER> 75 <ENTER>
```

sets `fill-column` to 75.

`<Alt+X> set-variable` is limited to user options, customizable variables, but you can set any variable with an elisp expression like this:

```
(setq fill-column 75)
```

To execute such an expression, type `<Alt+:>` (eval-expression) and enter the expression in the minibuffer. Alternatively, go to the `*scratch*` buffer, type in the expression, and then type `<Ctrl+J>`.

Setting variables this way affects only the current emacs session. The only way to alter the variable for future sessions is to put the alteration as a Lisp statement in your initialization file.

3.3.13.3 Init File Syntax

Your GNU emacs system's *init* file, **~/.emacs**, contains elisp expressions. Each elisp expression consists of a function name followed by arguments, all surrounded by parentheses. For example:

```
(setq fill-column 60)
```

calls the function `setq` to set the variable `fill-column` to 60.

You can set any Lisp variable with `setq`, but with certain variables `setq` won't work.

The second argument to `setq` is an expression for the new value of the variable. This can be a constant, a variable, or a function call expression. In your **~/.emacs** file, constants are used most of the time. They can be:

Numbers: Numbers are written in decimal, with an optional initial minus sign.

Strings: Lisp string syntax is the same as C string syntax with a few extra features. Use a double-quote character (`"`) to begin and end a string constant.

Characters: Lisp character constant syntax consists of a `?` followed by either a character or an escape sequence starting with `\`.

True: `t` stands for "true."

False: `nil` stands for "false."

Other Lisp objects: Write a single quote (`'`) followed by the Lisp object you want.

3.3.13.4 Keystroke Abbreviations or Abbrevs

Just like an ordinary language abbreviation, an *abbrev* is a word which expands, when you insert it, into a usually expanded or enlarged string of text. Abbrevs are defined by the user to expand in specific ways. For example, you might define `Bob` as an abbrev expanding to `Better off built`. Then you could insert `Better off built` into the buffer by typing `Bob <Space>`.

A second kind of abbreviation facility, which we do not show examples of here, is called *dynamic abbrev expansion*. You use dynamic abbrev expansion with an explicit command to expand the letters in the buffer before the point by looking for other words in the buffer that start with those letters.

Abbrevs expand only when *Abbrev mode*, a *buffer-local minor mode*, is enabled. Disabling Abbrev mode does not cause abbrev definitions to be forgotten, but they do not expand until Abbrev mode is enabled again. The command `<Alt+X> abbrev-mode` toggles Abbrev mode; with a numeric argument, it turns Abbrev mode on if the argument is positive, off otherwise.

You can define abbrevs interactively during the editing session, irrespective of whether

Abbrev mode is enabled. You can also save lists of abbrev definitions in files, which you can then reload for use in later sessions.

3.3.13.4.1 Defining Abbrevs: The following are ways of defining and managing abbrevs: **`<Ctrl+X> a g`**

Define an abbrev, using one or more words before point as its expansion (`add-global-abbrev`).

`<Ctrl+X> a l`

Similar, but define an abbrev specific to the current major mode (`add-mode-abbrev`).

`<Ctrl+X> a i g`

Define a word in the buffer as an abbrev (`inverse-add-global-abbrev`).

`<Ctrl+X> a i l`

Define a word in the buffer as a mode-specific abbrev (`inverse-add-mode-abbrev`).

`<Alt+X> define-global-abbrev <Enter> abbrev <Enter> expression <Enter>`

Define abbrev as an abbrev expanding into an `expression`.

`<Alt+X> define-mode-abbrev <Enter> abbrev <Enter> expression <Enter>`

Define abbrev as a mode-specific abbrev expanding into an `expression`.

`<Alt+X> kill-all-abbrevs`

Discard all abbrev definitions, leaving a blank slate.

The usual way to define an abbrev is to enter the text you want the abbrev to expand to, position the point after it, and type `<Ctrl+X> a g`. This reads the abbrev itself using the minibuffer, and then defines it as an abbrev for one or more words before the point. As with many other emacs commands, you can use a numeric digit argument to specify how many words before the point should be taken as the expansion. For example, to define the abbrev `Bob`, insert the text `Better off built` and then type `<Ctrl+U> 3 <Ctrl+X> a g Bob <Enter>`.

An argument of zero to `<Ctrl+X> a g` means to use the contents of the region as the expansion of the abbrev being defined.

To remove an abbrev definition, give a negative argument to the abbrev definition command, as in one of the following ways:

`<Ctrl+U> - <Ctrl+X> a g`

`<Ctrl+U> - <Ctrl+X> a l`

The first way removes a global definition, while the second way removes a mode-specific definition.

`<Alt+X> kill-all-abbrevs` removes all abbrev definitions, both global and local.

3.3.13.4.2 Controlling Abbrev Expansion: When Abbrev mode is enabled, an abbrev expands whenever it is present in the buffer just before the point and when you type a self-inserting whitespace or punctuation character like `<Space>` or a comma, etc. More precisely, any character that is not a word constituent expands an abbrev, and any word constituent character can be part of an abbrev. The most common way to use an abbrev is to insert it and then insert a punctuation or whitespace character to expand it.

These commands are used to control abbrev expansion.

`<Alt+'>`

Separate a prefix from a following abbrev to be expanded (`abbrev-prefixmark`).

`<Ctrl+X> a e`

Expand the abbrev before the point (`expand-abbrev`). This is effective even when Abbrev mode is not enabled.

`<Alt+X> expand-region-abbrevs`

Expand some or all abbrevs found in the region.

If you expand an abbrev by mistake, you can undo the expansion by typing `C-/` (undo). This undoes the insertion of the abbrev expansion and brings back the abbrev text. You can also use the command `<Alt+X> unexpand-abbrev` to cancel the last expansion without deleting the terminating character.

3.3.13.4.3 Listing and Editing Abbrevs: **`<Alt+X> list-abbrevs`**

Display a list of all abbrev definitions. With a numeric argument, list only local abbrevs.

`<Alt+X> edit-abbrevs` allows you to add, change or kill abbrev definitions by editing a list of them in an emacs buffer. The buffer of abbrevs is called ***Abbrevs***, and is in **Edit>Abbrevs mode**. Type `<Ctrl+C> <Ctrl+C>` in this buffer to install the abbrev definitions as specified in the buffer, and delete any abbrev definitions not listed.

The commands `edit-abbrevs` and `list-abbrevs` are the same except they display the listing in a window and a buffer, respectively.

3.3.13.4.5 Saving Abbrevs: These commands allow you to keep abbrev definitions between editing sessions:

`<Alt+X> write-abbrev-file <Enter> filename <Enter>`

Save to filename describing all defined abbrevs.

`<Alt+X> read-abbrev-file <Enter> filename <Enter>`

Read from **filename** and define abbrevs as specified in that file.

`<Alt+X> define-abbrevs`

Define abbrevs from definitions in current buffer.

`<Alt+X> insert-abbrevs`

Insert all abbrevs and their expansions into current buffer.

`<Alt+X> write-abbrev-file` reads a file name using the minibuffer and then writes a description of all current abbrev definitions into that file. This is used to save abbrev definitions for use in a later session. The text stored in the file is a series of Lisp expressions that, when executed, define the same abbrevs that you currently have.

`<Alt+X> read-abbrev-file` reads a file name using the minibuffer and then reads the file, defining abbrevs according to the contents of the file. The function `quietly-read-abbrev-file` is similar except that it does not display a message in the echo area; you cannot invoke it interactively, and it is used primarily in your init file. If either of these functions is called with `nil` as the argument, it uses the file given by the variable `abbrev-file-name`, which is **~/.emacs.d/abbrev_defs** by default. This is your standard abbrev definition file, and Emacs loads abbrevs from it automatically when it starts up.

Emacs will offer to save abbrevs automatically if you have changed any of them, whenever it offers to save all files (for `<Ctrl+X> s` or `<Ctrl+X> <Ctrl+C >`). It saves them in the file specified by `abbrev-file-name`. This feature can be inhibited by setting the variable `save-abbrevs` to `nil`.

The commands `<Alt+X> insert-abbrevs` and `<Alt+X> define-abbrevs` are similar to the previous commands but work on text in an emacs buffer. `<Alt+X> insert-abbrevs` inserts text into the current buffer after the point, describing all current abbrev definitions; `<Alt+X> define-abbrevs` parses the entire current buffer and defines abbrevs accordingly.

3.3.13.5 Keystroke Macro Commands

Similar to the brief introduction in Section 3.3.5, in this section we more fully describe how to record, save, edit, and list a sequence of commands in a *macro*, so you can repeat it conveniently later. A keyboard macro is a command defined by an emacs user that represents, in a shortened form, another sequence of keys. For example, if you discover that you are about to type three different keystroke combinations 400 times, you can speed your work by defining a much shorter keyboard macro to do those three different keystroke combinations and then executing it 399 more times.

You define a keyboard macro by executing and recording the commands which are its definition. As you define a keyboard macro, the definition is being executed for the first time. When you close the definition, the keyboard macro is defined and also has been executed once. You can then repeat the commands by invoking the macro as many times as you like.

3.3.13.5.1 Keystroke Macros: Basic Use These are the basic operations in defining and using keystroke macros.

<F3>

Start defining a keyboard macro (`kmacro-start-macro-or-insert-counter`).

<F4>

Dual-purpose function key. If a keyboard macro is being defined, end the definition; otherwise, execute the most recent keyboard macro (`kmacro-end-or-call-macro`).

<Ctrl+U> <F3>

Re-execute last keyboard macro, then append keys to its definition.

<Ctrl+U> <Ctrl+U> <F3>

Append keys to the last keyboard macro without re-executing it.

<Ctrl+X> <Ctrl+K> r

Run the last keyboard macro on each line that begins in the region (apply-macro-to-region-lines).

To start defining a keyboard macro, type `<F3>`. From then on, your keys continue to be executed, but also become part of the definition of the macro. `Def` appears in the mode line. When you are finished, type `<F4>` (`kmacro-end-or-call-macro`) to terminate the definition. For example:

```
<F3> <Alt+F> Mansoor <F4>
```

defines a macro to move forward a word and then insert `Mansoor` at the point. `<F3>` and `<F4>` do not become part of the macro.

After defining the macro, it is the most recently defined keyboard macro, and you can call it with `<F4>`. In the example, this has the same effect as typing `<Alt+F> Mansoor` again.

The two roles of the `<F4>` command: it ends the macro if you are in the process of defining one, or calls the last macro otherwise.

You can also supply `<F4>` with a numeric prefix argument `n`, which means to invoke the macro `n` times. An argument of zero repeats the macro indefinitely, until it gets an error or you type `<Ctrl+G>` to terminate it.

After ending the definition of a keyboard macro, you can append more keystrokes to its definition by typing `<Ctrl+U> <F3>`. This is equivalent to plain `<F3>` followed by retyping the whole definition so far. As a consequence, it re-executes the macro as previously defined. If you change the variable `kmacro-execute-before-append` to `nil`, the existing macro will not be re-executed before appending to it (the default is `t`). You can also add to the end of the definition of the last keyboard macro without re-executing it by typing `<Ctrl+U> <Ctrl+U> <F3>`.

When a command reads an argument with the minibuffer, your minibuffer input becomes part of the macro along with the command. So when you replay the macro, the command

gets the same argument as when you entered the macro. For example:

`<F3> <Ctrl+A> <Ctrl+K> <Ctrl+X> b Mansoor <Enter> <Ctrl+Y> <Ctrl+X> b <Enter> <F4>`

defines a macro that kills the current line, yanks it into the buffer **Mansoor**, then returns to the original buffer. The command **`<Ctrl+X> <Ctrl+K> r`** (`apply-macro-to-region-lines`) repeats the last defined keyboard macro on each line that begins in the region. It does this line by line, by moving the point to the beginning of the line and then executing the macro.

All defined keyboard macros are recorded in the *keyboard macro ring*. There is only one keyboard macro ring, shared by all buffers. The basic keyboard macro ring operations are

`<Ctrl+X> <Ctrl+K> <Ctrl+K>`

Execute the keyboard macro at the head of the ring (`kmacro-end-or-callmacro-repeat`).

`<Ctrl+X> <Ctrl+K> <Ctrl+N>`

Rotate the keyboard macro ring to the next macro (defined earlier) (`kmacrocycle-ring-next`).

`<Ctrl+X> <Ctrl+K> <Ctrl+P>`

Rotate the keyboard macro ring to the previous macro (defined later) (`kmacrocycle-ring-previous`).

Note: The maximum number of macros stored in the keyboard macro ring is determined by the customizable variable `kmacro-ring-max`.

3.3.13.5.2 Naming, Saving, and Invoking or Using Keyboard Macros The following are the ways to name, save, and invoke or use keyboard macros, particularly with regard to retaining them in your **~/.emacs** or init file so that they will be available in all future sessions of emacs (anything below enclosed in `[ ]` is optional).

1. **`<Ctrl+X> <Ctrl+K> n <Enter> macroname <Enter>`**

   Gives a command name (for the duration of the current emacs session only) to the most recently defined keyboard macro (`kmacro-name-last-macro`). If you wish to save a keyboard macro for later use, you can give it a name using this syntax. This sequence reads a name as an argument, by prompting for the name in the minibuffer, and uses the minibuffer-supplied name and defines that name so that you can execute the last keyboard macro, in its current form, using that name. The macro name is an elisp symbol, and defining it in this way makes it a valid command name for invoking or using it with `<Alt+X>`, or for binding a key to it with `global-set-key`. If you specify a name that has a prior definition other than a keyboard macro, you get an error.

2. **`<Ctrl+X> <Ctrl+K> b <Enter> key <Enter>`**

   Binds the most recently defined keyboard macro to a key sequence (for the duration of the current emacs session only) (`kmacro-bind-to-key`).

3. **`<Alt+X> insert-kbd-macro <Enter> [macroname <Enter>]`**

   Inserts in the current buffer a keyboard macro's definition as elisp code. If you do not supply an already-defined `macroname`, the last keyboard macro defined is inserted as elisp code.

4. **`<Alt+X> macroname <Enter>`**

   Invokes `macroname` in the current buffer.

5. Pressing the function key `<F4>` invokes the last defined keyboard macro.

3.3.13.5.3 Saving Keyboard Macros for Future Sessions Once a keyboard macro has a name, you can save its definition in a file, and particularly in the **~/.emacs** file or other initialization file that you may use to initialize emacs at startup. By taking the following steps, it can be used in all future editing sessions.

The steps to accomplish this are as follows:

1. Visit the file you want to save the definition in, which becomes the current buffer. This is usually **~/.emacs**.

2. Use the command `<Alt+X> insert-kbd-macro <Enter> macroname <Enter>`

   This uses the `macroname` you already have previously defined, and inserts equivalent elisp code that the keyboard macro represents, into the current buffer.

3. Save the current buffer. If the file you save in is your initialization file **~/.emacs**, then the macro will be defined for all future sessions of emacs.

3.3.13.6 Redefining Keyboard Keys

This section describes key bindings, which map keys to commands, and *keymaps*, which record key bindings. It also explains how to customize key bindings, which is done by editing your init file.

3.3.13.5.4 Keys, Commands, and Variables Emacs does not assign meanings to keys directly. Instead, emacs assigns meanings to named commands, and then gives keys their meanings by binding them to commands.

As you have seen in the previous sections, every command has a name, which is usually made up of a few words separated by hyphens—for example, `insert-kbd-macro` or `abbrev-file-name`. Internally, each command is an emacs form of a Lisp function, and the actions associated with the command are performed by running the function.

The bindings, or mappings, between keys and commands are recorded in tables called *keymaps*.

The effect of "`<Ctrl+N>` moves point down vertically one line" is that the vertical movement of the command `next-line` is bound to the key sequence `<Ctrl+N>`. If you rebind `<Ctrl+N>` to the command `forward-word`, `<Ctrl+N>` will move forward one word instead. The key is bound to a command.

A *variable* is a name used to store a value. The variables we described in Section 3.3.13.2 are intended to be customized: some commands or mechanisms in emacs examine the variable and behave according to the value that you assign to the variable when and if you customize it.

3.3.13.5.5 Keymaps Emacs commands are elisp functions whose definition provides for interactive use. Like every elisp function, a command has a function name, which usually consists of lowercase letters and hyphens. A keystroke (*key* for short) sequence is a sequence of input events that have a meaning as a unit. Input events include characters, function keys, and mouse buttons—all the inputs that you can send to the computer. A key sequence gets its meaning from its binding, which dictates what command it runs.

The bindings between key sequences and command functions are recorded in data structures called *keymaps*. Emacs has many of these, each used on particular occasions.

The global keymap is the most important keymap because it is always in effect. The *global keymap* defines keys for *Fundamental mode*; most of these definitions are common to most or all major modes. Each major or minor mode can have its own keymap which overrides the global definitions of some keys.

For example, a self-inserting character such as `g` is self-inserting because the global keymap binds it to the command `self-insert-command`. The standard emacs editing characters such as `<Ctrl+A>` also get their standard meanings from the global keymap. Commands to rebind keys, such as `<Alt+X> global-set-key`, work by storing the new binding in the proper place in the global map.

Most modern keyboards have function keys as well as character keys. Function keys send input events just as character keys do, and keymaps can have bindings for them. Key sequences can mix function keys and characters. For example, if your keyboard has a `<Home>` function key, emacs can recognize key sequences like `<Ctrl+X> <Home>`. You can even mix mouse events with keyboard events, such as `S-down-mouse-1`.

On text terminals, typing a function key actually sends the computer a sequence of characters; the precise details of the sequence depends on the function key and on the terminal type. (Often the sequence starts with `ESC [`). If emacs understands your terminal type properly, it automatically handles such sequences as single input events.

3.3.13.5.6 Prefix Keymaps Emacs stores only single events in each keymap. Interpreting a key sequence of multiple events involves a chain of keymaps: the first keymap gives a definition for the first event, which is another keymap that is used to look up the second event in the sequence, and so on. A *prefix key* such as `<Ctrl+X>` or `<Esc>` has its own keymap, which holds the definition for the event that immediately follows that prefix.

A prefix key is usually the keymap to use for looking up the following event. The definition can also be an elisp symbol whose function definition is the following keymap; the effect is the same, but it provides a command name for the prefix key that can be used as a description of what the prefix key is for. Thus, the binding of `<Ctrl+X>` is the symbol `Control-X-prefix`, whose function definition is the keymap for `<Ctrl+X>` commands. The definitions of `<Ctrl+C>`, `<Ctrl+X>`, `<Ctrl+H>`, and `<Esc>` as prefix keys appear in the global map, so these prefix keys are always available.

Some prefix keymaps are stored in variables with names.

`ctl-x-map` is the variable name for the map used for characters that follow `<Ctrl+X>`.

_ **`help-map`** is for characters that follow `<Ctrl+H>`.

_ **`esc-map`** is for characters that follow `<Esc>`. Thus, all metacharacters are actually defined by this map.

_ **`ctl-x-4-map`** is for characters that follow `<Ctrl+X> 4`.

_ **`mode-specific-map`** is for characters that follow `<Ctrl+C>`.

3.3.13.5.7 Local Keymaps So far, we have explained the ins and outs of the global map. Major modes customize emacs by providing their own key bindings in *local keymaps*. For example, C mode overrides `<Tab>` to make it indent the current line for C code. Minor modes can also have local keymaps; whenever a minor mode is in effect, the definitions in its keymap override both the major mode's local keymap and the global keymap. In addition, portions of text in the buffer can specify their own keymaps, which override all other keymaps.

A local keymap can redefine a key as a prefix key by defining it as a prefix keymap. If the key is also defined globally as a prefix, its local and global definitions (both keymaps) effectively are combined: both definitions are used to look up the event that follows the prefix key. For example, if a local keymap defines `<Ctrl+C>` as a prefix keymap, and that keymap defines `<Ctrl+Z>` as a command, this provides a local meaning for `<Ctrl+C>` `<Ctrl+Z>`. This does not affect other sequences that start with `<Ctrl+C>`; if those sequences don't have their own local bindings, their global bindings remain in effect.

Another way to think of this is that emacs handles a multievent key sequence by looking in several keymaps, one by one, for a binding of the whole key sequence. First it checks the minor mode keymaps for minor modes that are enabled, then it checks the major mode's keymap, and then it checks the global keymap.

3.3.13.5.8 Changing Key Bindings Interactively The way to redefine an emacs key is to change its entry in a keymap. You can change the global keymap, in which case the change is effective in all major modes (except those that have their own overriding local bindings for the same key), or you can change a local keymap, which affects all buffers using the same major mode.

The following describes how to rebind keys for the current emacs session (see Section 3.3.13.5.9 for a description of how to make key rebindings affect future emacs sessions by putting them in your **~/.emacs** file):

1. **`<Alt+X> global-set-key <Enter> key command <Enter>`**

   Defines `key` globally to run `command`.

2. **`<Alt+X> local-set-key <Enter> key command <Enter>`**

   Defines `key` locally (in the major mode now in effect) to run `command`.

3. **`<Alt+X> global-unset-key <Enter> key`**

   Makes `key` undefined in the global map.

4. **`<Alt+X> local-unset-key <Enter> key`**

Makes `key` undefined locally (in the major mode now in effect).

For example, the following binds `<Ctrl+Z>` to the shell command, replacing the normal global definition of `<Ctrl+Z>`:

```
<Alt+X> global-set-key <Enter> <Ctrl+Z> shell <Enter>
```

The `global-set-key` command reads the command name after the key. After you press the key, a message like this appears so that you can confirm that you are binding the key you want:

```
Set key <Ctrl+Z> to command:
```

You can redefine function keys and mouse events in the same way; just type the function key or click the mouse when it's time to specify the key to rebind. You can rebind a key that contains more than one event in the same way. Emacs keeps reading the key to rebind until it is a complete key (that is, not a prefix key). Thus, if you type `<Ctrl+F>` for the key, that's the end; it enters the minibuffer immediately to read the command. But if you type `<Ctrl+X>`, since that's a prefix, it reads another character; if that is `4`, another prefix character, it reads one more character, and so on. For example:

```
<Alt+X> global-set-key <Enter> <Ctrl+X> 4 $ spell-other-win-
  dow <Enter>
```

redefines `<Ctrl+X> 4 $` to run the (fictitious) command `spell-other-window`.

You can remove the global definition of a key with `global-unset-key`. This makes the key undefined; if you type it, emacs will just beep. Similarly, `local-unset-key` makes a key undefined in the current major mode keymap, which makes the global definition (or lack of one) come back into effect in that major mode.

If you have redefined (or undefined) a key and you subsequently wish to retract the change, undefining the key will not do the job; you need to redefine the key with its standard definition.

To find the name of the standard definition of a key, go to a Fundamental mode buffer in an emacs session that you have not done any key remappings in, and type `<Ctrl+H> c`. So, if you want to prevent yourself from invoking a command by mistake, it is better to disable the command than to undefine the key!

3.3.13.5.9 Rebinding Keys in Your Init File If you have a set of key bindings that you like to use all the time, you can specify them in your initialization file by writing elisp code. There are several ways to write a key binding using elisp. The simplest is to use the `kbd` function, which converts a text representation of a key sequence, similar to how we have written key sequences up to this point, into a form that can be passed as an argument to `global-set-key`. For example, here's how to bind `<Ctrl+Z>` to the `shell` command.

```
(global-set-key (kbd "C-z") 'shell)
```

The single quote (`'`) before the `shell` command name designates it as a constant symbol rather than a variable. If you omit the quote, emacs tries to evaluate `shell` as a variable.

3.3.13.5.10 Examples Here are some additional examples, including binding function keys and mouse events:

```
(global-set-key (kbd "<Ctrl+C> y") 'clipboard-yank)

(global-set-key (kbd "<Ctrl ><Alt+Q>") 'query-replace)

(global-set-key (kbd "<f5>") 'flyspell-mode)

(global-set-key (kbd "<Ctrl ><f5>") 'linum-mode)

(global-set-key (kbd "<Ctrl ><right>") 'forward-sentence)

(global-set-key (kbd "<mouse-2>") 'mouse-save-then-kill)
```

EXERCISE 3.11

a. Use the emacs Help function via keyboard keystrokes only to find out what the commands that are being bound to each of the keys sequences in the six examples accomplish. So, for `forward-sentence`, what explanation does Help supply? Make a list of the answers that the Help function supplies.

b. What are the default key sequence bindings, if any, for the commands in the six examples? Make a list of the default key sequence bindings for commands that have them.

EXERCISE 3.12

Place all six examples of key sequences bound to commands in your **~/.emacs** file and test them according to your findings in Exercise 3.10.

3.4 vi AND EMACS COMMAND TABLES (TABLES 3.17 AND 3.18)

3.5 SUMMARY

In this chapter, we covered vi/vim and emacs, the two most useful families of text editors that UNIX offers. We achieved this in both a command line, text-based way, and in a graphical way, for both families of editor. The editors are useful because modern UNIX is both a text-driven and GUI-based operating system. Common operations done by an ordinary user, such as editing script files, writing e-mail messages, or creating C language programs, are done with text editors. A full-screen display editor shows a portion of a file that fills most or all of the screen display. The cursor, or point, can be moved to any of the text shown in the screen display. Editing a file involves editing a copy that the editor creates, called a buffer. Keystroke commands are one of the primary ways of interacting with these editors. Using a GUI to interact with both families of editor is time efficient and easy to learn. The editor(s) used should fit the user's personal criteria.

We showed how to modify any of several environment options to customize the behavior of the vi, vim, and gvim editors, either when you are in the editor for one session only or for every editor session. These vi, vim, and gvim user options include specifying maximum line length and automatically wrapping the cursor to the next line, displaying line numbers as you edit a file, and displaying the mode that the editor is in. We showed how to use full or abbreviated names for most of the options.

We described the basic methods of customizing and modifying the behavior of GNU emacs. This included the following customization procedure operations: using the emacs Options menu to modify options; using Custom (a GUI-based interface) to change preferences and options, and in conjunction with that interface, also using the traditional typed `<Alt+X> customize` command set; writing keystroke abbreviations with Abbrev; writing keystroke macro commands; how to redefine keyboard keys; and placing emacs Lisp (elisp) code to customize the behavior of emacs directly into your **~/.emacs** startup configuration file.

The most important functions that are common to these UNIX text editors are cursor movement, cut/copy and paste, deleting text, inserting text, opening an existing file, starting a new file, quitting, saving, and search and replace.

TABLE 3.17 vi, vim, gvim Summary

Vi Syntax

Command	**Action**
`cw`	Change word.
`cc`	Change line.
`c$`	Change text from current position to end of line.
`C`	Same as `c$`.
`dd`	Delete current line.
`7 dd`	Delete seven lines.
`d$`	Delete text from current position to end of line.
`D`	Same as `d$`.
`5dw`	Deletes five words.
`d7,14`	Deletes lines seven through fourteen in the buffer.
`s`	Substitute character. `<Esc>` ends substitute mode.
`4s`	Substitute four characters. `<Esc>` ends substitute mode.
`S`	Substitute entire line. `<Esc>` ends substitute mode.
`u`	Undo last change.
`<Ctrl+R>`	Redo last change (vim and gvim).
`U`	Restores the current line, if you have not moved off of it.
`x`	Delete current cursor position.
`X`	Delete back one character.
`5X`	Delete previous five characters
`.`	Repeat last change.
`~`	Change case and move cursor right.
`<Ctrl+A>`	Increment number at the cursor (vim and gvim).
`<Ctrl+X>`	Decrement number at the cursor (vim and gvim).

Vi Mode Keys

Key	**Action**
`a`	Appends text after the character the cursor is on.
`A`	Appends text after the last character of the current line.
`c`	Begins a change operation, allowing you to modify text.
`C`	Changes from the cursor position to the end of the current line.
`i`	Inserts text before the character the cursor is on.
`I`	Inserts text at the beginning of the current line.
`o`	Opens a blank line below the current line and puts the cursor on that line.
`O`	Opens a blank line above the current line and puts the cursor on that line.

Vi Command Mode

Command	**Action**
`:wq`	Saves the buffer and quits.
`:w`	Saves the current buffer and remains in the editor.
`:w filename`	Saves the current buffer to filename
`:q`	Quit vi (fails if changes were made).
`:q!`	Quit vi without saving the buffer.
`:Q`	Quit vi and invoke ex.
`:vi`	Return to vi after Q command.
`ZZ`	Quits vi, saving the file only if changes were made since the last save.

TABLE 3.17 (Continued) vi, vim, gvim Summary

Vi Cursor Movement	
Command	**Action**
`1G`	Moves the cursor to the first line of the file.
`G`	Moves the cursor to the last line of the file.
`0 (zero)`	Moves the cursor to the first character of the current line.
`<Ctrl+G>`	Reports the position of the cursor in terms of line # and column #.
`$`	Moves the cursor to the last character of the current line.
`w`	Moves the cursor forward one word at a time.
`b`	Moves the cursor backward one word at a time.
`x`	Deletes the character at the cursor position.
`dd`	Deletes the line at the current cursor position.
`u`	Undoes the most recent change.
`r`	Replaces the character at the current cursor location with what is typed next.
Vi Yank and Put	
Command Syntax	What It Accomplishes
`y2W`	Yanks two words, starting at the current cursor position, going to the right.
`4yb`	Yanks four words, starting at the current cursor position, going to the left.
`yy or Y`	Yanks the current line.
`p`	Puts the yanked text after the current cursor position.
`P`	Puts the yanked text before the current cursor position.
Vi Substitute	
Command Syntax	**What It Accomplishes**
`:s/john/jane/`	Substitutes the word `jane` for the word `john` on the current line, only once.
`:s/john/jane/g`	Substitutes the word `jane` for every word `john` on the current line.
Vi Environment Options	
Last Line Mode Syntax	**What it does**
abbr command	
`:ab in out`	Use `in` as abbreviation for `out` in Insert mode.
`:unab in`	Remove abbreviation for `in`.
`:ab`	List abbreviations.
`map!, map commands`	
`:map`	List character strings that are mapped.
`:map! string sequence`	Map characters string to input mode sequence.
`:unmap! string`	Remove input mode map (you may need to quote the characters with `<Ctrl+V>`).
`:map!`	List character strings that are mapped for input mode.
set command	
`:set x`	Enable boolean option `x`, show value of other options.
`:set`	Show changed options.
`:set all`	Show all options.
`:set x?`	Show value of option `x`.

TABLE 3.18 emacs Summary

Emacs Commands

Command	**Action**
`<Ctrl+X> <Ctrl+F>`	Visit a file (`find-file`)
`<Ctrl+X> <Ctrl+R>`	Visit a file for viewing, without allowing changes to it (`find-file-read-only`)
`<Ctrl+X> <Ctrl+V>`	Visit a different file instead of the one visited last (`find-alternate-file`)
`<Ctrl+X> <Ctrl+S>`	Save the current buffer to its file (`save-buffer`)
`<Ctrl+X> s`	Save any or all buffers to their files (`save-some-buffers`)
`<Alt+~>`	Forget that the current buffer has been changed (`not-modified`)
`<Ctrl+X> <Ctrl+W>`	Save the current buffer with a specified file name (`write-file`)
`<Ctrl+H>`	Display a help message about these options
`<Ctrl+X> <Ctrl+C>`	Exits emacs
`<Ctrl+X> <Ctrl+Z>`	Suspends emacs and exits to the shell
Emacs Help Command	
`<Ctrl+H> a topics <Enter>`	Display a list of commands whose names match `topics` (`apropos-command`).
`<Ctrl+H> b`	Display all active key bindings—minor mode bindings first, then those of the major mode, then global bindings (`describe-bindings`).
`<Ctrl+H> c key`	Show the name of the command that the key sequence `key` is bound to (`describe-key-briefly`). Here `c` stands for "character." For more extensive information on `key`, use `<Ctrl+H> k`.
`<Ctrl+H> d topics <Enter>`	Display the commands and variables whose documentation matches `topics` (`apropos-documentation`).
`<Ctrl+H> e`	Display the ***Messages*** buffer (`view-echo-area-messages`).
`<Ctrl+H> f function <Enter>`	Display documentation on the Lisp function named `function` (`describe-function`). Since commands are Lisp functions, this works for commands too.
`<Ctrl+H> r`	Display the emacs manual in Info (`info-emacs-manual`).
`<Ctrl+H> s`	Display the contents of the current syntax table (`describe-syntax`). The syntax table says which characters are opening delimiters, which are parts of words, and so on.
`<Ctrl+H> t`	Enter the emacs interactive tutorial (help-with-tutorial).
`<Ctrl+H> K key`	Enter Info and go to the node that documents the key sequence `key` (`Info-goto-emacs-key-command-node`).
`<Ctrl+H>`	Display the help message for a special text area, if the point is in one (`display-local-help`). (These include, for example, links in ***Help*** buffers.)

TABLE 3.18 (Continued) emacs Summary

Emacs Cursor Movement

Entity to Move Over		**Backward**	**Forward**
Character		`<Ctrl+B>`	`<Ctrl+F>`
Word		`<Alt+B>`	`<Alt+F>`
Line		`<Ctrl+P>`	`<Ctrl+N>`
Go to line beginning (or end)		`<Ctrl+A>`	`<Ctrl+E>`
Sentence		`<Alt+A>`	`<Alt+E>`
Paragraph		`<Alt+{>`	`<Alt+}>`
Page		`<Ctrl+X> [`	`<Ctrl+X> ]`
Entity to Kill		**Backward**	**Forward**
Character (delete, not kill)		`<Del>`	`<Ctrl+D>`
Word		`<Alt+Del>`	`<Alt+D>`
Line (to end of)		`<Alt+0> <Ctrl+K>`	`<Ctrl+K>`
Sentence		`<Ctrl+X> DEL`	`<Alt+K>`
Kill region	`<Ctrl+W>`		
Copy region to kill ring	`<Alt+W>`		
Yank back last thing killed	`<Ctrl+Y>`		

Emacs Interactive Search and Replace

Search and Replace Action	**Keystrokes**
Search forward	`<Ctrl+S>`
Search backward	`<Ctrl+R>`
Regular expression search	`<Ctrl+Alt+S>`
Reverse regular expression search	`<Ctrl+Alt+R>`
Select previous search string	`<Alt+P>`
Select next later search string	`<Alt+N>`
Exit incremental search	`<Enter>`
Undo effect of last character	`<Del>`
Abort current search	`<Ctrl+G>`
Interactively replace a text string	`<Alt+%>`
Using regular expressions	`<Alt+X> query-replace-regexp`
Replace this one, go on to next	`<Space> or y`
Replace this one, don't move	`,`
Skip to next without replacing	`<Del> or n`
Replace all remaining matches	`!`
Back up to the previous match	`^`
Exit query-replace	`<Enter>`
Enter recursive edit (<**Ctrl+Alt+C**> to exit)	`<Ctrl+R>`

(*Continued*)

TABLE 3.18 (Continued) emacs Summary

Changing Emacs Behavior Customization Action	Keystrokes
Abbrevs	
add global abbrev	`<Ctrl+X> a g`
add mode-local abbrev	`<Ctrl+X> a l`
add global expansion for this abbrev	`<Ctrl+X> a i g`
add mode-local expansion for this abbrev	`<Ctrl+X> a i l`
explicitly expand abbrev	`<Ctrl+X> a e`
expand previous word dynamically	`<Alt+/>`
Macros	
Start defining a keyboard macro	`<Ctrl+X> (` or <F3>
End keyboard macro definition	`<Ctrl+X> )` or <F4>
Execute last-defined keyboard macro	`<Ctrl+X> e` or <F4>
Append to last keyboard macro	`<Ctrl+U> <Ctrl+X> (`
Name last keyboard macro	`<Alt+X> name-last -kbd-macro`
Insert Lisp definition in buffer	`<Alt+X> insert-kbd-macro`
Customize variables and faces	`<Alt+X> customize`

QUESTIONS AND PROBLEMS

Vi, Vim, Gvim

1. Despite the availability of fancy and powerful word processors, why is text editing still important?
2. List 10 commonly used text-editing operations.
3. What are the four most popular text editors in UNIX? Which one is your favorite? Why?
4. What is an editor buffer?
5. This problem assumes you are using the C shell on PC-BSD, and will execute the file you will create from your home directory on the system. If you are running Solaris, make the appropriate changes in the following **sheller** script file to make it work in Bash. Be aware that there is no **/etc/shells** file in Solaris, so you can omit that line from your **sheller** script file.

   a. How do you make sure that your search path includes the directory you are saving the following script file to, and that you have execute privileges on the file?

   Use vi on your system and create a C shell script file that contains the lines:

   ```
   #!/bin/csh
   ```

```
echo $SHELL
cat /etc/shells
```

Then save the file as **sheller** and quit vi. At the C shell prompt, type `sheller` and then press `<Enter>`.

b. What appears on your screen? In particular, what shells are available?

6. Run vi on your system. Create and edit a block of text that you want to be the body of an e-mail message explaining the basic capabilities of the vi editor. For example, part of your message might describe the difference between the Insert and Command modes. This file should be at least one page (45 to 50 lines of text) long. Then save the file as **vi_doc.txt**. Insert the body of text in an e-mail message and send it to yourself.
7. Run vi on your system and create a file of definitions in your own words, without looking at the textbook, for:

   ```
   full-screen display editor
   modeless editor
   file versus buffer
   keystroke commands
   substitute versus search
   text file versus binary file
   ```

   Then refer back to the relevant sections of this chapter to check your definitions. Make any necessary corrections or additions. Re-edit the file in vim to incorporate any corrections or additions that you made, and then print out the file using the print commands available on your system.
8. Edit the file you created in Problem 7, and change the order of the text of your definitions to (d), (a), (c), and (b), using the `yank`, `put`, and `D` or `dd` commands. Print out the file using the print commands available on your system.
9. In Section 3.2.7, you changed the behavior of vi, vim, and gvim by adding or modifying entries in your **~/.exrc** or **~/.vimrc** files, so that the changes were persistent across all sessions of the editors. You can also customize vi, vim, and gvim by changing the shell environment variable named `EXINIT`. This can be achieved in the C shell by giving value(s) to the `SETENV` variable. Do the following:

   a. Refer to Chapter 14 for the C shell to find the exact syntax and use of the `SETENV` command in the C shell. Then, add or modify the shell environment variable setting for the shell variable `EXINIT` so that the `showmode` user option is turned on. What is the syntax of the command you used to do this?

b. How would you test that this environment variable is actually implementing the user option change, and not what is in the **~/.exrc** or **~/.vimrc** files?

c. What syntax would you use for the `SETENV` command to change more than one user option in the editors?

d. Are these changes in the `EXINIT` variable persistent through all vi, vim, and gvim sessions? If you log out and log back in to the system, does `EXINIT` still contain the changes and additions you made to it? Why or why not?

10. Give the exact syntax of a vi `substitute` command line that only replaces every instance of the discrete word `ate` on all the lines of a file with the word `ion`, where the file has some words that end in the string `ate`.

11. Give the exact syntax of a vim `substitute` command line that interactively searches and substitutes the word `cool` for the word `cold` on all the lines of a file, where there are several widely separated instances of the word `cold` in the file.

12. Take the following `map` command for creating a skeleton C program template, and place it in your **~/.exrc** file:

```
map #3 ^[i#include stdio.h ^Mmain(argc, argv) ^M int argc;^M
char #argv[];^M{^M}^M^[
```

where:

`^[` stands for pressing `<Ctrl+v>` and then `<Esc>`

`^M` stands for pressing `<Ctrl+v>` and then the `<Enter>` key

As stated in the text, the relative number of spaces in the `map` command definition controls the indentation of the skeleton construct. Also, the `^M` entries put each of the skeleton construct components on a new line.

a. Make sure that the relative indentation of the header components and other parts of the skeleton is correct.

b. Add `#include <stderr.h>`, `#include <stdlib.h>`, and `#include <string.h>` as header information to the skeleton.

c. Run the `map` command in a blank vi buffer and test it.

13. To practice with the command line window to reuse any previous searches in your search history, use vim to edit the file multiline you created in Practice Session 3.4. Then search for the words `engineers`, `system administrators`, `web servers`, `scientists`, `networking`, and `mathematicians`, one search at a time, starting at the first character in the buffer. Open a command line window on your search history. Modify the search commands in the history of searches for `scientists`, `networking`, and `mathematicians` to be commands that substitute

the words `people`, `homeserver gurus`, and `UNIX students` for the words `scientists`, `networking`, and `mathematicians`. Save the file.

GNU Emacs

14. This problem assumes that you are running PC-BSD and you can interactively start up a new shell, the Bourne Again shell, or *Bash*, which is already installed by default on that system. To do this, at the C shell prompt just type `bash` and press `<Enter>`. If a **~/.bashrc** exists, before you begin, be sure to back up your existing **~/.bashrc** file by using the `cp` command. To do so, type `cp .bashrc .bashrc_bak` and then press `<Enter>`. If for any reason you destroy the contents of the **~/.bashrc** file while doing this problem, you can restore the original by typing `cp .bashrc_bak .bashrc` and then pressing `<Enter>`.

    This problem is not meant for Solaris.

    If there is no **.bashrc** file in your home directory, use emacs to create one and save it in your home directory as an empty file (with nothing in it). Also, type `chmod u+X .bashrc` and press `<Enter>`.

    Use emacs to edit the **~/.bashrc** file in your home directory, and then use the `<Ctrl+x> I` command to insert the file **alien3** that you created in Practice Session 3.12 into the buffer. Save the buffer, exit emacs, and log off your computer system. Log on to your computer system again, start up a new Bash shell interactively by typing `bash` at the command line (so that the new **~/.bashrc** is in effect), and test each of the DOS aliases that are in **alien3** by typing them at the shell prompt, with their proper arguments (if necessary). They should give you the same results as when you ran the Bourne shell aliases in Step 21 of Practice Session 3.12.

    What other way can you invoke the **~/.bashrc** file immediately in this interactive session without logging off the system?

15. As you saw in the practice sessions, you can be editing more than one file at a time in emacs, where each of the files' contents are being held in different buffers. Experiment by first using the `cp` command at the shell prompt to make a copy of the file datafile that you created in Practice Session 3.9. Name this copy **datafile2**. Use emacs to open both files, **datafile** and **datafile2**, with the command `<Ctrl+x><Ctrl+f>`. You can switch between buffers with `<Ctrl+x> b`. Then edit both of them at the same time and cut and paste three or four lines of each between the two, using `<Ctrl+@>`, `<Ctrl+w>`, and `<Ctrl+y>`.

    Don't save your changes to the file datafile!

16. Write a keyboard macro, as described in Section 3.3.13.4, to do everything shown in Steps 10–16 of Practice Session 3.10.

17. Try working with emacs in a text-only window, and use only keystroke commands.

To do this, you will have to launch emacs from a console or terminal window by typing `emacs  -nw  newfile`. The `-nw` option specifies that emacs will run in nongraphical mode. Then, in the console or terminal window, a nongraphical emacs will open on the buffer **newfile**. As stated in Section 3.3, you can still gain access to the menu bar menus at the top of the emacs screen by pressing the escape `<Esc>` key on the keyboard and then pressing the single back quote (`) key. You can then descend through the menu bar choices by pressing the letter key of the menu choice you want to make. For example, pressing the `f` key on the keyboard gives you access to the File pull-down menu choices, and then pressing the `s` key on the keyboard allows you to save the current buffer.

18. To compare keystroke to graphical emacs, repeat Problem 15, using purely graphical emacs—that is, with no keystroke commands allowed. This time, make two copies of datafile named **datafilex** and **datafilexx** at the UNIX shell prompt with the `cp` command. Open all three files and, using the multiple-buffer and multiple-window capabilities of an X Window emacs, cut and paste among the files using only the mouse. Again, as in Problem 15, don't save your changes to the file datafile.
19. Use emacs's capability of sending e-mail while you're in emacs. Send an e-mail message to one of your friends, composing the message body and sending from within emacs.
20. Use the `<Alt+X> customize` facility in emacs to find the values of the following: Global Mark Ring Max, Tab Width, Fill Column, Standard Indent, Undo Limit, and provide a list of the values you find for each.
21. What emacs command toggles Abbrev mode? What emacs command removes all abbrev definitions, including global ones?
22. Define the following abbreviations as global abbreviations in emacs with Abbrev using the word on the left of the equal sign (=) as the abbreviation, and list the commands and keystrokes you used to create the abbreviations, and invoke them:

    now = Now is the time for all good women to come to the aid of their country.

    BSD = PC-BSD

    SUN = Oracle Solaris
23. Define a GNU emacs keyboard macro that, when invoked, automatically enters all 26 lowercase letters of the alphabet, with a single space between each letter, at the point. Name the macro `le` and bind it to the key `1` (the numeric number 1) for use only during this session of emacs. Give the exact steps, commands, and typed input you use to accomplish defining this macro and invoking it.
24. Define a GNU emacs keyboard macro that, when invoked, automatically enters the integers 1 through 10, with a single space in between each number, at the point. Name that macro `row` and bind it to the key `r` so that both the name and the key binding

can be used in every subsequent emacs session. Give the exact steps, commands, and typed input you use to accomplish defining this macro and invoking it.

25. Define a line of elisp code that will designate the second mouse button on your mouse to issue a command to split the current buffer window horizontally and place it in your ~/**.emacs** file.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

CHAPTER 4

Files and File System Structure

Objectives

- To explain briefly the UNIX file system structure and the ZFS file system
- To explain the UNIX file concept
- To discuss various types of files supported by UNIX
- To describe attributes of a file
- To explain the notion of pathnames
- To explain the user view of the UNIX file system
- To describe the user interface to the UNIX file system—browsing the file system
- To discuss representation of a file inside the UNIX system
- To describe how a UNIX file is stored on the disk
- To explain the concept of standard files in UNIX
- To cover the commands and primitives

  `~`, `.`, `..`, `/`, *`PATH`*, `cd`, `echo`, `file`, `getconf`, `ls`, `mkdir`, `pwd`, `rmdir`

4.1 INTRODUCTION

Most computer users work with the file system structure of the computer system that they use. While using a computer system, a user is constantly performing file-related operations: creating, reading, writing/modifying, or executing files. Therefore, the user needs to understand what a file is in UNIX, how files can be organized and managed, how they are represented inside the operating system, and how they are stored on the disk. In this chapter, the description of file representation and storage is simplified, due to the scope of this textbook. More details on these topics are available in books on operating system concepts and principles and in books on UNIX internals.

4.2 THE UNIX FILE CONCEPT

One of the many remarkable features of the UNIX operating system is its concept of files. This concept is simple, yet powerful, and results in a uniform view of all system resources. In UNIX, a file is a sequence of bytes. Period. Thus, everything, including a network interface card, a disk drive, a USB flash drive, a keyboard, a printer, a simple/ordinary (text, executable, etc.) file, or a directory, is treated as a file. As a result, all input and output devices are treated as files in UNIX, as described under file types and file system structure.

4.3 TYPES OF FILES

UNIX supports seven types of files:

- Simple/ordinary file
- Directory
- Symbolic (soft) link
- Character special file
- Block special file
- Named pipe (FIFO)
- Socket

You can use the `ls -l` command to display the type of a file, as shown in Table 4.3.

4.3.1 Simple/Ordinary File

Simple/ordinary files are used to store information and data on a secondary storage device, typically a disk. An ordinary file can contain a source program in C, C++, C#, Java, PHP, ROR, Python, Perl, LISP, and so on; an executable program that you have created by compiling (and linking) a source program or applications such as compilers, database tools, desktop publishing tools, graphing software, and so on; PostScript code, pictures, audio, graphics, and so on. UNIX does not treat any one of these files differently from another. It does not give a structure or attach a meaning to a file's contents, because every file is simply a sequence of bytes. Meanings are attached to a file's contents by the application that uses/processes the file. For example, a C program file is no different to UNIX from an HTML file for a Web page or a file for a video clip. However, a C compiler (e.g., cc or gcc), a Web browser (e.g., Firefox), and a video player (e.g., RealPlayer) treat these files differently.

You can name files by following any convention that you choose to use; UNIX does not impose any naming conventions on files of any type. File names can have as many as 14 letters in System V and 255 letters in Berkeley Software Distributions (BSD). Most contemporary UNIX systems comply with the BSD naming scheme. You can use the `getconf` command on your PC-BSD system to display the maximum size of a file name in number of characters, as follows:

```
$ getconf _XOPEN_NAME_MAX
255
$
```

If you get an error message, then you can use the `getconf NAME _ MAX` command.

Although you can use any characters for file names, we strongly recommend that nonprintable characters, white spaces (spaces and tabs), and shell metacharacters (described in Chapter 2) not be used because they are difficult to deal with as part of a file name. You can give file names any of your own or application-defined extensions, but the extensions mean nothing to the UNIX system. For example, you can give an **.exe** extension to a document and a **.doc** extension to an executable program. Some applications require extensions, but others do not. For example, all C compilers require that C source program files have a **.c** extension, but not all Web browsers require an **.html** extension for files for Web pages. Even so, extensions should be used—it helps keep track of which files are for what purposes. Some commonly used extensions are given in Table 4.1.

4.3.2 Directory

A directory contains the names of other files and/or directories (the terms directory and subdirectory are used interchangeably). In some systems, terms such as *folder*, *drawer*, or *cabinet* are used for a directory. A directory file in any operating system consists of an array of directory entries, although the contents of a directory entry vary from one system to another. In UNIX, a directory entry has the structure shown in Figure 4.1.

The *inode number* is four bytes long and is an index value for an array on the disk. An element of this array, known as an index node, more commonly called an *inode*, contains the attributes of a file such as file size (in bytes). When you create a new file, the UNIX kernel allocates an inode to it. Thus, every unique file in UNIX has a unique inode (and

TABLE 4.1 Commonly Used Extensions for Some Applications

Extension	Contents of File
.bmp, .jpg, jpeg, .gif	Graphics
.c	C Source code
.C, .cpp, .cc	C++ Source code
.java	Java source code
.class	Java class file
.html, .htm	File for a Web page
.o	Object code
.ps	Postscript code
.Z, .gz	Compressed

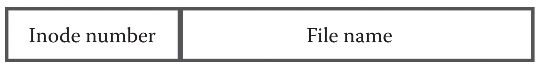


FIGURE 4.1 Structure of a directory entry.

inode number). The details of an inode and how the kernel uses it to access a file's contents on disk are discussed in Section 4.6.

4.3.3 Link File

A file of type link "points to" an existing file. The content of a link file is the pathname of the existing file. Thus, a link file allows you to access an existing file through another path in the file system structure and share it without duplicating its contents. The concept of a link in UNIX is fully discussed in Chapter 8. But, for now, a file of type link is created by the system when a *symbolic link* is created to an existing file. Symbolic links are a creation of BSD UNIX but are currently available on almost all versions of UNIX.

4.3.4 Special (Device) File

A *special file*, also known as a *device node*, is a means of accessing hardware devices, including the keyboard, disks, tape drive, graphic cards, network cards, and printers. Each hardware device is associated with at least one special file—and a command or an application accesses a special file to access the corresponding device. Special files are divided into two types: *character special files* and *block special files*. Character special files correspond to character-oriented devices, such as a keyboard, and block special files correspond to block-oriented devices, such as a disk. Special files are typically placed in the **/dev** directory (see Section 4.4 for more details).

Applications and commands read and write peripheral device files in the same way that they read or write an ordinary file. That capability is the main reason that input and output in UNIX is said to be *device independent*. Various special devices simulate physical devices and are, therefore, known as *pseudodevices*. These devices allow you to interact with a UNIX system without using the devices that are physically connected to it. These devices are becoming more and more important, because they allow use of a UNIX system via a network or modem, or with virtual terminals in a window system such as the X Window System (see Chapter 23).

4.3.5 Named Pipe (FIFO)

UNIX has several tools that enable processes to communicate with each other. These tools, which are the key to the ubiquitous client–server software paradigm, are called *interprocess communication (IPC) mechanisms* (commonly known as *IPC primitives* or *IPC channels*). These primitives are called *pipes*, *named pipes* (also called *FIFOs*), and *sockets* (systems that are strictly System V–compliant have a mechanism called *transport layer interface* [TLI]). These primitives are discussed in detail under "System Programming" in Chapters 18 through 21. Here, we briefly mention the purpose of each so that you can appreciate the need for each mechanism and understand the need for FIFOs.

A pipe is an area in the kernel memory (a kernel buffer) that allows two processes to communicate with each other, provided the processes are running on the same computer system and are related to each other; typically, the relationship is parent–child or sibling. A FIFO is a file (of named pipe type) that allows two processes to communicate with each other if the processes are on the same computer; however, the processes do not have to be related to each other through a parent–child or sibling relationship. We illustrate the use of pipes and FIFOs at the command level in Chapter 9.

4.3.6 Socket

A socket can be used by processes on the same computer or on different computers to communicate with each other; the computers can be on a network (intranet) or on the Internet. Sockets can belong to different address families, each specifying the protocol suite to be used by processes to communicate. For example, the application layer protocols such as the HyperText Transfer Protocol (HTTP) use sockets of address family **AF_INET** in the TCP/IP protocol suite for communication (see Chapter 11 for a detailed discussion on TCP/IP). A socket with address family **AF_INET** is also known as the *Internet domain socket*, which means that processes running on computers on the Internet can use sockets of this domain to communicate with each other. A socket with address family **AF_UNIX** can be used for communication between processes that run on the same machine under a UNIX operating system. This kind of socket is also known as a *UNIX domain socket*. On a System V UNIX computer, a socket file type means a UNIX domain socket.

4.4 FILE SYSTEM STRUCTURE

Three issues are related to the file system structure of an operating system. The first is how files in the system are organized from the user's point of view. The second is how files are stored on the secondary storage (usually, a hard disk). The third is how files are manipulated (read, written, etc.). In this chapter, we will address the first issue, that is, the user view of files and directories in a UNIX system. As has been the case so far, our focus will be on PC-BSD UNIX.

4.4.1 File System Organization

The UNIX file system is structured hierarchically and is treelike, but upside-down, with the root at the top. Thus, the file system structure starts with one main directory, called the root directory, and can have any number of files and subdirectories under it, organized in any desired way. This structure leads to a parent–child relationship between a directory and its subdirectories/files. A typical UNIX system contains hundreds of files and directories. For our PC-BSD system, the files and directories under the root directory, denoted as / in UNIX terminology, are shown in the following session:

```
$ ls -l /
total 46
-r--r--r--   1 root  root  6142 Feb 25 01:20 COPYRIGHT
drwxr-xr-x   2 root  root    47 Feb 25 01:19 bin
```

```
drwxr-xr-x   8 root  root    43 Aug 14 14:43 boot
drwxr-xr-x   3 root  root     3 Jul  9 19:00 compat
drwxr-xr-x   3 root  root     3 Jul  9 19:10 data
dr-xr-xr-x   8 root  root   512 Aug 14 14:43 dev
-rw-------   1 root  root  4096 Aug 14 14:43 entropy
drwxr-xr-x  21 root  root   116 Aug  8 09:42 etc
lrwxr-xr-x   1 root  root     9 Jul  9 19:05 home -> /usr/home
drwxr-xr-x   3 root  root    50 Feb 25 01:20 lib
drwxr-xr-x   3 root  root     5 Jul  9 18:57 libexec
drwxr-xr-x   2 root  root     2 Feb 25 01:19 media
drwxr-xr-x   2 root  root     2 Feb 25 01:19 mnt
dr-xr-xr-x   1 root  root     0 Aug 16 13:44 proc
drwxr-xr-x   2 root  root   142 Feb 25 01:20 rescue
drwxr-xr-x   7 root  root    17 Jul  9 14:33 root
drwxr-xr-x   2 root  root   134 Jul  9 19:05 sbin
lrwxr-xr-x   1 root  root    11 Feb 25 01:21 sys -> usr/src/sys
drwxrwxrwt  22 root  root    37 Aug 16 09:46 tmp
drwxr-xr-x  20 root  root    20 Jul  9 19:10 usr
drwxr-xr-x  25 root  root    25 Aug 14 14:43 var
$
```

Note that the root directory contains 46 files and directories, with 42 directories, two ordinary files, and two symbolic links (see Chapter 8). In addition, there are the current directory (**.**) and parent of the current directory (**..**). You can display the long listing of all the files, including directories as well as the other dot files using the `–al` options. The big-picture user view of our system's files and directories is shown in Figure 4.2.

4.4.2 Home and Present Working Directories

When you log on, the UNIX system puts you in a specific directory, called your home/login directory. For example, the directory called **sarwar** in Figure 4.2 is the home directory for the user with the login **sarwar**. While using the C, tcsh, Bash, or Korn shell, you can specify your home directory by using the tilde (~) character. The directory that you are in at any particular time is called your *present working directory* (also known as your *current directory*). The present working directory is also denoted as **.** (pronounced "dot"). The parent of the present working directory is denoted as **..** (pronounced "dot dot").

Later in this chapter, we describe the commands you can use to determine your home and present working directories. We also identify commands you can use to interact with the UNIX file system in general.

4.4.3 Pathnames: Absolute and Relative

A file or directory in a hierarchical file system is specified by a *pathname*. Pathnames can be specified in three ways: (1) starting with the root directory, (2) starting with the present working directory, and (3) starting with the user's home directory. When a pathname is specified starting with the root directory, it is called an *absolute pathname* because it

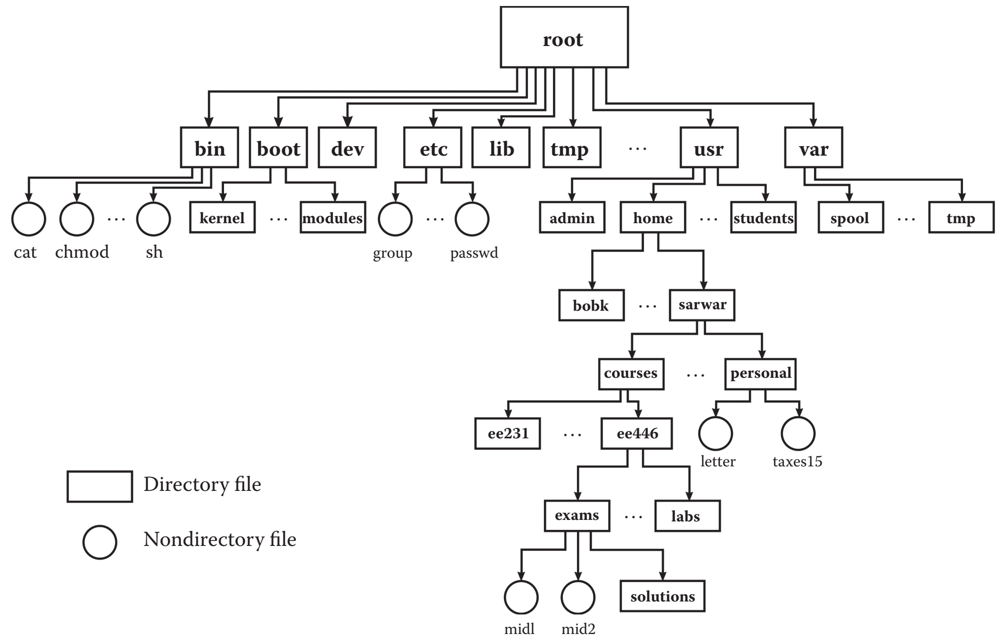


FIGURE 4.2 Typical UNIX file structure.

can be used by any user from anywhere in the file system structure. For example, **/home/faculty/sarwar/courses/ee446** is the absolute pathname for the **ee446** directory under the user **sarwar's** home directory. The absolute pathname for the file called **mid1** under **sarwar's** home directory is **/home/faculty/sarwar/courses/ee446/exams/mid1**.

Pathnames starting with the present working directory or a user's home directory are called **relative pathnames**. When the user **sarwar** logs on, the system puts him into his home directory, **/home/faculty/sarwar**. While in his home directory, **sarwar** can specify the file **mid1** (see Figure 4.2) by using a relative pathname, **./courses/ee446/exams/mid1** or **courses/ee446/exams/mid1**. The user **sarwar** (or anyone else) in the directory **ee446** can specify the same file with the relative pathname **exams/mid1**. The owner (or anyone logged on as the owner) of the **mid1** file can also specify it from anywhere in the file structure by using the pathname **~/courses/ee446/exams/mid1** or **$PATH/courses/ee446/exams/mid1**. Or, you could specify **ee446** from your personal directory as **../courses/ee446**.

A typical UNIX system has several disk drives that contain user and system files, but, as a user, you do not have to worry about which disk drive contains the file that you need to access. In UNIX, multiple disk drives and/or disk partitions can be *mounted* on the same file system structure, allowing their access as directories and not as named drives A:, B:, C:, and so on, as in MS-DOS and Microsoft Windows. You can access files and directories on these disks and/or partitions by specifying their pathnames as if they are part of the file structure on one disk/partition. Doing so gives a unified view of all the files and directories in the system, and you do not have to worry about remembering the names of drives and the files and directories they contain.

4.4.4 Some Standard Directories and Files

Every UNIX system contains a set of standard files and directories. The standard directories contain some specific files. In this section, we discuss some of the important directories. You may like to browse through the first website listed in Web Resources (Table 4.4) to know more about the FreeBSD (and PC-BSD) directory hierarchy.

Root directory (/): The root directory is at the top of the file system hierarchy and is denoted as a slash (/). It contains some standard files and directories and, in a sense, is the master cabinet that contains all drawers, folders, and files.

/bin: Also known as the binary directory, the **/bin** directory contains binary (i.e., executable) images of most UNIX programs/commands that are fundamental to starting and repairing single-user and multiuser environments. These commands include `cat`, `chmod`, `cp`, `csh`, `date`, `echo`, `kill`, `ln`, `ls`, `mkdir`, `mv`, `pgrep`, `ps`, `pwd`, `rm`, `rmdir`, `sh`, `stty`, `sync`, `tar`, `tcsh`, `test`, and `unlink`. The superuser and ordinary users may use these programs.

On BSD, the **/usr/bin** directory is different from **/bin**. On Solaris, the **/usr** and **/usr/bin** directories are one and the same thing because **/bin** is a symbolic link to **/usr/bin**. On BSD, the **/usr/bin** directory contains hundreds of executable programs that are not needed for starting or repairing the system. A few of the most commonly used are `CC`, `awk`, `cal`, `clear`, `diff`, `du`, `env`, `file`, `find`, `finger`, `gdb`, `gunzip`, `gzip`, `head`, `join`, `last`, `less`, `locate`, `man`, `nl`, `pstree`, `quota`, `spell`, `ssh`, `sudo`, `tail`, `time`, `top`, `uniq`, `vi`, `w`, `wc`, `which`, `who`, `whoami`, `zcat`, `zdiff`, and `zgrep`. On our BSD, **/usr/bin** contains 486 files and on Solaris it contains 1205 files.

/boot: This directory contains the programs and configuration files that are used during the bootstrap process of your system. The **/boot/loader** file is the system loader, that is, the program that actually loads the kernel into the memory and make it runnable. The **/boot/defaults** directory contains the standard configuration files for bootstrapping. On some systems, it may only contain the **loader.conf** file. The **/boot/kernel** directory contains over 1500 kernel executable modules to be loaded into the memory at boot time. The **/boot/modules** directory contains third-party modules that can be loaded into the kernel at runtime.

/dev: The **/dev** directory, which is also known as the device directory, contains files corresponding to the devices connected to the computer, including terminals, disk drives, CD-ROM drive, tape drives, modems, graphics cards, network cards, printers, and so on. These files, called special files, were described in Section 4.3.4.

This directory contains at least one file for every device connected to the computer. Each device has a name and a number, and the special file representing the device reflects both. Some example files in the **/dev** directory are as follows: **ada0** is the first serial advanced technology attachment (SATA) hard drive, **kbd0** represents the keyboard, **cd0** is for compact disc 0, **cdrom** for CD-ROM (which is a symbolic link to **cd0**, as is **dvd**), **pp0** represents a parallel port, **lpt0** is for a printer, **tty**'s for (teletype) terminals, and **usb** for Universal Serial Bus ports. The **/dev/pts** directory is used to manage pseudoterminals.

A system may have several devices of each type—for example, 10 hard disks or partitions, 20 terminals, 100 pseudoterminals, two solid state disks (SSDs), and so on. Our PC-BSD based system contains a total of 124 files in the **/dev** directory. This directory may contain several hundred—even over 1000—files in a network-based UNIX environment in a medium-to-large-sized organization.

/etc: The **/etc** directory contains commands, files, and scripts needed for system configuration and administration. A typical user is generally not allowed to use the commands and files in this directory. Some of the files and directories in this directory include **crontab**, **csh.cshrc**, **csh.login**, **csh.logout**, **group**, **inetd.conf**, **login.access**, **login.conf**, **passwd**, **printcap**, **profile**, **rc.d**, **rcp**, **shells**, **services**, **ssh**, **ssl**, and **termcap**. Discussion of most of the files in this directory is beyond the scope of this textbook. However, we briefly discuss the **/etc/passwd** file toward the end of this section. We discuss a few more files in this directory under "System Programming" in Chapters 18 through 21 and in Chapter 22 on "System Administration."

/lib: The library directory contains a collection of related files for a given language in a single file called an archive. A typical UNIX system today contains libraries for C and C++. The archive file for one of these languages can be used by applications developed in that language. The **/lib** directories contains libraries that are critical for the executable programs in the **/bin** and **/sbin** directories. The **/usr/lib** directory contains the shared and archive-type libraries (created by the `ar` command).

/tmp: Used by several commands and applications, the **/tmp** directory contains temporary files. You can use this directory for your own temporary files as well. All the files in this directory are deleted periodically so that the disk (or a partition of the disk) does not get filled with temporary files. The life of a file in the **/tmp** directory is set by the system administrator and varies from system to system, but it is usually only a few minutes. Files in **/tmp** may or may not exist when a system is rebooted.

/usr: The **/usr** directory contains subdirectories that hold, among other things, most of the utilities, system daemons (see Chapter 10), applications, programming tools, standard C include files, shared and archive-type language libraries, manual pages and other important documents, and source code (BSD and third-party). Two of the most important subdirectories in this directory are **bin** and **lib**, which contain binary images of most UNIX commands (utilities, tools, etc.) and language libraries, respectively.

/usr/home: Organized in some fashion, the **/usr/home** directory is normally used to hold the home directories of all the users of the system. For example, the system administrator can create subdirectories under this directory that contain home directories for certain types of users. For instance, the diagram in Figure 4.2, which shows a university-like setup, has one subdirectory each for the home directories of members of the administration, faculty, staff, students, and so on. These subdirectories are labeled **admin**, **faculty**, and **students**. Our system administrator created a symbolic link (see Chapter 8) called **/home** that points to **/usr/home**, as shown in the following session.

```
% ls -ld /home
lrwxr-xr-x 1 root root 9 Jul 9 19:05 /home -> /usr/home
%
```

This means that if a user, say **john**, changes the directory to his/her home directory and runs the pwd command, the command output would be **/usr/home/john** and not **/home/john**. We discuss links in UNIX, including symbolic links, in Chapter 8.

/var: The **/var** directory contains multipurpose log, temporary, and spool files. Among several other directories, the **/var/mail** directory contains files for receiving and holding incoming e-mail messages of users. When you read your new e-mail, it comes from a file in this directory. The **/var/spool/mqueue** directory contains the undelivered mail queue and the **/var/spool/output** directory contains the line printer spooling directories. The **/var/tmp** directory contains temporary files that are kept between system reboots.

/etc/passwd: The **/etc/passwd** file contains one line for every user on the system and describes that user. Each line has seven fields, separated by colons. The following is the format of the line.

```
login_name:password:user_ID:group_ID:user_info:home_
directory:login_shell
```

The login _ name is the login name by which the user is known to the system and is what the user types to log in. The password field contains the dummy password x or * in newer systems (starting with System V Release 4—SVR4) and the encrypted version of the password in older systems. The newer versions store encrypted passwords in the **/etc/master.passwd** file. Only the superuser (i.e., **root**) has read and write access permissions for the master password file; nobody else can even read it. POSIX requires user _ ID (UID) to be an integer type. Usually, the superuser is assigned a UID of **0**. Several other login names are also assigned UIDs that are known (or from a known range). Typically, UIDs 1–499 or 1–999 are reserved. Depending on the UNIX system that you use, UIDs 1,000–32,767 or 1,000–65,536 are assigned to "normal" users like you and I. In systems that use 32-bit UIDs, this range is 1,000–4,294,967,296. The group _ ID identifies the group that the user belongs to, and it also is an integer between 0 and 65,535 with, usually, integers 0–99 reserved. The user _ info field contains information about the user, typically the user's full name. The home _ directory field contains the absolute pathname for the user's home directory. The last field, login _ shell, contains the absolute pathname for the user's login shell. The command corresponding to the pathname specified in this field is executed by the system when the user logs on. Back-to-back colons mean that the field value is missing, which is sometimes done with the user _ info field. The following session shows the line from the **/etc/passwd** file on our system for the user **sarwar**:

```
% cat /etc/passwd | grep "sarwar"
sarwar:*:1004:1008:Mansoor Sarwar:/home/sarwar:/bin/csh
%
```

In this line, the login name is **sarwar**, the password field contains *, the user ID is 1004, the group ID is 1008, the personal information is the user's full name (Mansoor Sarwar), the home directory is **/home/sarwar**, and the login shell is **/bin/csh**, or the C shell. We usually work under the Bourne shell by running the /bin/sh command after login.

The following in-chapter exercises give you practice in browsing the file system on your UNIX machine and help you understand the format of the **/etc/passwd** file.

EXERCISE 4.1

Go to the **/dev** directory on your system and identify one character special file and one block special file.

EXERCISE 4.2

View the **/etc/passwd** file on your system to determine your user ID.

4.5 NAVIGATING THE FILE STRUCTURE

Now, we describe some useful commands for browsing the UNIX file system, creating files and directories, determining file attributes, determining the absolute pathname for your home directory, determining the pathname for the present working directory, and determining the type of a file. The discussion is based on the file structure shown in Figure 4.2 and the user name **sarwar**.

4.5.1 Determining the Absolute Pathname for Your Home Directory

When you log on, the system puts you in your home directory. You can find the full pathname for your home directory by using the `echo` and `pwd` commands.

With no argument, the `echo` command displays a blank line on the screen. You can determine the absolute pathname of your home directory by using the `echo` command, as follows:

```
$ echo $HOME
/home/sarwar
$
```

where *HOME* is a shell variable (a placeholder) in the Bourne shell. The shell uses this variable to keep track of the absolute pathname of your home directory. In the C shell, the variable is `home`. We discuss shell variables and the `echo` command in detail in Chapters 12 through 15.

Another way to display the absolute pathname of your home directory is to use the `pwd` command. You use this command to determine the absolute pathname of the directory you are currently in, that is, your present working directory, also known as the current directory. This command does not require any arguments. When you log on, the UNIX system puts you in your home directory. You can use the `pwd` command right after logging on to display the absolute pathname of your home directory, as follows:

```
$ pwd
/home/sarwar
$
```

If you are using the C shell, the same command would display **/usr/home/sarwar** as the absolute pathname of your home directory. However, the `ls –ld` commands in the following session with **/usr/home/sarwar** and **/home/sarwar** as arguments produce the same result, as follows:

```
% pwd
/usr/home/sarwar
% ls -ld /usr/home/sarwar
drwxr-xr-x 26 sarwar faculty 47 Aug 15 11:49 /usr/home/sarwar
% ls -ld /home/sarwar
drwxr-xr-x 26 sarwar faculty 47 Aug 15 11:49 /home/sarwar
%
```

This is so because **/home** is a symbolic link to **/usr/home**.

4.5.2 Browsing the File System

You can browse the file system by going from your home directory to other directories in the file system structure and displaying a directory's contents (files and subdirectories in the directory), provided that you have the *permissions* to do so. We cover file security and access permissions in detail in Chapter 5. For now, we show how you can browse your own files and directories by using the `cd` (change directory) and `ls` (list directory) commands. The following is a brief description of the `cd` command.

SYNTAX

```
cd [directory]
```

Purpose: Change the present working directory to **directory**, or to the home directory if no argument is specified

The shell variable `PWD` is set after each execution of the `cd` command. The `pwd` command uses the value of this variable to display the present working directory. After getting into a directory, you can view its contents (the names of files or subdirectories in it) by using the `ls` command. The following is a brief description of this command. The `cd` and `ls` commands are two of the most heavily used UNIX commands.

SYNTAX

```
ls [option] [pathname-list]
```

Purpose: Send the names of the files in the directories and files specified in **pathname-list** to the display screen

Output: Names of the files and directories in the directory specified by **pathname-list**, or the names only if **pathname-list** contains file names only

Commonly used options/features:

`-F`	Display / after directories, * after binary executables, and @ after symbolic links
`-a`	Display names of all files, including hidden files **.**, **..**, and so on.
`-i`	Display inode number and file name
`-l`	Display long list that includes access permissions, hard link count, owner, group, file size (in bytes), and modification time

If the command is used without any argument, it displays the names of files and directories in the present working directory. The following session illustrates how the `ls` and `cd` commands work with and without parameters. The `pwd` command displays the absolute pathname of the current directory. With the exception of hidden files, the `ls` command displays the name of all the files and directories in the current directory. The `cd courses` command is used to make the **courses** directory the current directory. The `cd ee446/exams` command makes **ee446/exams** the current directory. The `ls ~` and `ls $HOME` commands display the names of the files and directories in your home directory. The `cd` command without any argument puts you in your home directory. In other words, it makes your home directory your current directory.

```
$ pwd
/home/sarwar
$ ls
Desktop   Downloads Images    Videos    personal
Documents GNUstep   Music     courses   unix3e
$ cd courses
$ ls
ee231   ee446
$ cd ee446/exams
$ pwd
/home/sarwar/courses/ee446/exams
$ ls
mid1 mid2
$ ls ~
Desktop   Downloads Images    Videos    personal
Documents GNUstep   Music     courses   unix3e
$ ls $HOME
Desktop   Downloads Images    Videos    personal
Documents GNUstep   Music     courses   unix3e
$ cd
$ ls
Desktop   Downloads Images    Videos    personal
Documents GNUstep   Music     courses   unix3e
$
```

We demonstrate the use of the `ls` command with various options in the remainder of this chapter and other chapters of the book. We use the terms *flag* and *option* interchangeably.

In a typical UNIX system, you are not allowed to access all the files and directories in the system. In particular, you are typically not allowed to access many important files and directories related to system administration and files and directories belonging to other users. However, you have permissions to read a number of directories and files. The following session illustrates that we have permissions to go to and list the contents of, among many other directories, the / and **/usr** directories.

```
$ cd /usr
$ ls
bin    home    jails   lib32   libexec obj    ports   share   swap
games  include lib     libdata local   pbi    sbin    src     tests
$ cd /
$ ls
COPYRIGHT compat    entropy   lib       mnt       root      tmp
bin       data      etc       libexec   proc      sbin      usr
boot      dev       home      media     rescue    sys       var
$ cd
$ ls /usr
bin    home    jails   lib32   libexec obj    ports   share   swap
games  include lib     libdata local   pbi    sbin    src     tests
$
```

Without any option, the `ls` command does not show all the files and directories; in particular, it does not display the names of hidden files. Examples of these files include **.**, **..**, **.bash_history**, **.bashrc**, **.config**, **.cshrc**, **.history**, **.login**, **.mailrc**, **.profile**, **.rhosts**, **.shrc**, **.ssh**, and **.xsession**. We have already discussed the **.** and **..** directories. The purposes of some of the more important hidden files are summarized in Table 4.2.

You can also display the names of all the files and directories in a directory, including the hidden files, by using the `ls` command with the –a option. In the following session, the `cd` command places you in your home directory and the `ls –a` command displays all the files and directories, including the hidden files, in your home directory.

```
$ cd
$ ls -a
.              .gnome2        .lesshst       .shrc          Downloads
..             .gtkrc-2.0     .login         .ssh           GNUstep
.bash_history  .history       .login_conf    .windowlab     Images
.cinnamon      .i3            .mail_aliases  .xprofile      Music
.config        .icewm         .mailrc        .xscreensaver  Videos
.cshrc         .icons         .profile       .xsession      courses
.fluxbox       .ideskrc       .ratpoisonrc   .zshrc         personal
.fvwm-crystal  .idesktop      .rhosts        Desktop        unix3e
.gconf         .kde4          .screenrc      Documents
$
```

You can use shell *metacharacters* in specifying multiple files or directory parameters to the `ls` command. For example, the command `ls  /usr/*.c` displays the names of all

TABLE 4.2 Some Important Hidden Files and Their Purposes

File Name	Purpose
.	Present working directory
..	Parent of the present working directory
.bash_history	Contains the history of commands executed under `bash`
.bashrc	Setup for the Bash shell
.cshrc	Setup for the C shell
.exrc	Setup for vi
.login	Setup for shell if C or tcsh shells are the login shells; executed at login time
.mailrc	Setup and address book for `mail` and `mailx`
.profile	Setup for shell if Bourne or Korn shell is the login shell; executed at login time
.rhosts	Domain names of the trusted hosts (see Chapter 11 for details)
.shrc	Setup for shell if Bourne shell is the login shell; executed at login time
.ssh	Keys of the servers on which you would be allowed to login using the `ssh` command. Keys are stored when you try to establish the session on a server for the first time
.xsession	Customized X session script

C program files in the **/usr** directory. We discuss the use of metacharacters and *regular expressions* in detail in later in this chapter and in Chapters 6 and 7.

4.5.3 Creating Files

While working on a computer system, you need to create files and directories: files to store your work and directories to organize your files more efficiently. You can create files by using various tools and applications, such as editors, and create directories by using the `mkdir` command. In Chapter 3, we discussed text editors vim and emacs that you can use to create files containing plain text. You can create nontext files by using various applications and tools, such as a compiler, that translates source code in a high-level language (e.g., C) and generates a file that contains the corresponding executable code.

4.5.4 Creating and Removing Directories

We briefly discussed the `mkdir` and `rmdir` commands in Chapter 2. Here, we cover these commands fully. You can create a directory by using the `mkdir` command. The following is a brief description of this command.

SYNTAX

```
mkdir [option] directory-names
```

Purpose: Create directories specified in **directory-names**

Commonly used options/features:

`-m MODE`	Create directories with the access permissions specified in MODE in octal (see Chapter 5)
`-p`	Create parent directories that do not exist in the pathnames specified in **directory-names**

Here, **directory-names** are the pathnames of the directories to be created. When you log on, you can use the following command to create a subdirectory, called **memos**, in your home directory. Access permissions for the newly created directories are determined by the current value of `umask` (see Chapter 5). You can confirm the creation of this directory by using the `ls -ld memo` command, as in:

```
$ mkdir memos
$ ls -ld memo
drwxr-xr-x 2 sarwar faculty 68 Aug 15 07:42 memo
$
```

Similarly, you can create a directory called **test_example** in the **/tmp** directory by using:

```
$ mkdir /tmp/test_example
$
```

While in your home directory, you can create the directory **professional** and a subdirectory **letters** under it by using the `mkdir` command with the `-p` option, as in:

```
$ mkdir -p professional/letters
$
```

You can use the `rmdir` command to remove an empty directory. If a directory is not empty, you must remove the files and subdirectories in it before removing it. To remove nonempty directories, you need to use the `rm` command with the `-r` option (see Chapter 6). The following is a brief description of the `rmdir` command.

SYNTAX

`rmdir [option] directory-names`

Purpose: Remove the empty directories specified in **directory-names**
Commonly used options/features:

`-p` Remove empty parent directories also

The following command removes the **letters** directory from the present working directory. If **letters** is not empty, the `rmdir` command displays the error message `rmdir: letter: Directory not empty` on the screen. If **letters** is a file, the command displays the error message `rmdir: letters: Not a directory`.

```
$ rmdir letters
$
```

The following command removes the directory **letters** from your present working directory and **memos** from your home directory.

```
$ rmdir letters ~/memos
$
```

If the ~/**personal** directory contains only one subdirectory, called **diary**, and it is empty, you can use the following command to remove both directories.

```
$ rmdir -p ~/personal/diary
$
```

4.5.5 Determining File Attributes

You can determine the attributes of files by using the `ls` command with various options. The options can be used together, and their order does not matter. For example, you can use the `-l` option to get a long list of a directory that gives the attributes of files, such as the owner of the file, as follows:

```
$ ls -l
total 5
drwxr-x---   2 sarwar  faculty  512 Jan 23 09:37 courses
drwxr-----   2 sarwar  faculty   12 May 01 13:22 memos
drwx------   2 sarwar  faculty  163 May 05 23:13 personal
$ ls -l ~/courses/ee446/exams
-rwxr--r--   1 sarwar  faculty  1512 Mar 12 11:10 mid1
-rwxr--r--   1 sarwar  faculty  1485 May 19 14:34 mid2
drwxrwxrwx   2 sarwar  faculty    63 May 12 13:44 solutions
$
```

The information displayed by the `ls -l` command is summarized in Table 4.3.

In the preceding two uses of the `ls -l` command, **courses**, **memos**, **personal**, and **solutions** are directories, and **mid1** and **mid2** are ordinary files. As stated earlier, we discuss access permissions and user types in Chapter 5. The owner of the files is **sarwar**, who belongs to the group **faculty**. The values of the remaining fields are self-explanatory.

You can use the `ls` command with the `-i` option to display the inode numbers of files and directories. To display the inode number of a directory, you need to use the `ls –id` command. The following examples of its use show that the inode number for the **greeting** file is 6278611, and for the directories **courses**, **memos**, and **personal**, they are 6555603, 6555456, and 6555324, respectively.

```
$ ls -i greeting
6278611  greeting
```

```
$ ls -id courses memo personal
6555603  courses  6555456  memo  6555324  personal
$
```

The `ls -al` command displays the long list of all the files in a directory, as follows:

```
$ ls -al ~/courses/ee446/exams
total 66
drwxr-xr-x  26 sarwar  faculty   47 Aug 10 11:49 .
drwxr-xr-x  10 sarwar  faculty   10 Aug  8 09:42 ..
-rw-r--r--   1 sarwar  faculty   84 Aug 13 10:19 mid1
-rw-r--r--   1 sarwar  faculty   68 Aug 13 11:49 mid2
drwxr-xr-x   2 sarwar  faculty   12 Aug 14 22:55 solutions
$
```

You can use the `-F` option to identify directories, executable files, and symbolic links. The `ls -F` command displays an asterisk (*) after an executable file, a slash (/) after a directory, and an "at" symbol (@) after a symbolic link (discussed in Chapter 8), as follows:

```
$ ls -F /
COPYRIGHT  compat/   entropy   lib/       mnt/      root/     tmp/
bin/       data/     etc/      libexec/   proc/     sbin/     usr/
boot/      dev/      home@     media/     rescue/   sys@      var/
$
```

Note that there is no executable file in the root directory. The output of the `ls -F /bin` command would show that all the files in the **/bin** directory are executable. You are

TABLE 4.3 Summary of the Output of the `ls -l` Command (fields listed left to right)

Field	Meaning
First letter of first field	File type: - ordinary file b block special file c character special file d directory l link p named pipe (FIFO) s socket
Remaining letters of first field	Access permissions for owner, group, and others
Second field	Number of hard links
Third field	Owner's login name
Fourth field	Owner's group name (can also be a number)
Fifth field	File size in bytes
Sixth, seventh, and eighth field	Date and time of last modification
Ninth field	File name

encouraged to read the online manual pages for the `ls` command on your system, or see the Command Appendix at the CRC Press website for this book for a detailed description of the command.

By using the shell metacharacters and regular expressions, you can specify a particular set of files and directories in the file system structure, or a particular set of strings in files or directories. For example, the following command can be used in the C shell to display the long lists for all the files in the **~/courses/ee446** directory that have the **.c** extension and start with the string lab followed by zero or more characters, with the condition that the first of these characters cannot be 5.

```
$ ls -l ~/courses/ee446/lab[^5]*.c
...
$
```

Similarly, the following command can be used to display the inode numbers and names of all the files in your current directory that have four-character names and an **.html** extension. The file names must start with a letter, followed by any two characters, and end with a digit from 1 through 5.

```
$ ls -i [a-zA-Z]??[1-5].html
...
$
```

The following command under the C shell displays the names of all the files in your home directory that do not start with a digit and that end with **.c** or **.C**. In other words, the command displays the names of all the C and C++ source program files that do not start with a digit. Under the Bourne shell, you may replace the ^ character with the `!` character. Thus, the `ls ~/[!0-9]*.[c,C]` command would produce the same results.

```
$ ls ~/[^0-9]*.[c,C]
...
$
```

4.5.6 Determining the Type of a File's Contents

Because UNIX does not support file extensions, you can use any extension name for any file. This means that you can use the **.jpg** extension for an executable program file. Thus, you cannot determine the type of content of a file by simply looking at its name. Since many software tools require the use of extensions and the user may rely on extensions, extension names are, therefore, still significant. In UNIX, you can find the type of a file's contents by using the `file` command. Mostly, this command is used to determine whether a file contains text or binary data. Doing so is important because text files can be displayed on a terminal screen, whereas displaying the contents of a binary file shows "garbage" on your terminal screen and can also freeze your terminal, as it may interpret some of the binary values as control codes. The command has the following syntax.

SYNTAX

```
file [option] file-list
```

Purpose: Attempt to classify files in **file-list**
Commonly used options/features:
`-f FILE` Use FILE as a file of **file-list**

The following session shows a sample run of the command. In this case, the types of the contents of all the files in the root directory are displayed.

```
$ file /*
/COPYRIGHT: ASCII text
/bin:       directory
/boot:      directory
/compat:    directory
/data:      directory
/dev:       directory
/entropy:   regular file, no read permission
/etc:       directory
/home:      symbolic link to '/usr/home'
/lib:       directory
/libexec:   directory
/media:     directory
/mnt:       directory
/proc:      directory
/rescue:    directory
/root:      directory
/sbin:      directory
/sys:       broken symbolic link to 'usr/src/sys'
/tmp:       sticky directory
/usr:       directory
/var:       directory
$
```

The following session shows a few more types of files.

```
$ cd /bin
$ file cat freebsd-version rcp sh
cat:             ELF 64-bit LSB executable, x86-64, version 1
                 (FreeBSD), dynamically linked (uses shared libs),
                 for FreeBSD 10.0 (1000510), stripped
freebsd-version: POSIX shell script, UTF-8 Unicode text executable
rcp:             setuid ELF 64-bit LSB executable, x86-64, version
                 1 (FreeBSD), dynamically linked (uses shared
                 libs), for FreeBSD 10.0 (1000510), stripped
```

```
sh:                ELF 64-bit LSB executable, x86-64, version 1
                   (FreeBSD), dynamically linked (uses shared libs),
                   for FreeBSD 10.0 (1000510), stripped
$
```

The *executable and linkable format* (ELF) is a common standard file format for executable code, object code, shared libraries, and core dumps. UNIX creates a *core dump* in a file, called **core**, when a program crashes. Programmers can use this file to identify what caused the program to crash. It contains the program state at the time of crash: data that the crashed program was accessing at the time it crashed, the state of the program stack, and the location of the program statement that caused the crash. You can use the `file core` command to determine the name of the program that produced the core dump.

The **cat** and **sh** files contain ELF 64-bit executable codes, the **rcp** file contains set-UID ELF 64-bit executable code, and **freebsd-version** contains a POSIX shell script. Some more classifications that the `file` command displays are English text, C program text, Bourne shell script text, empty, nroff/troff, Perl command text, Python text, PostScript, sccs, and setgid executable. You should read the manual page for the command to learn more about the `file` command. The terms *SUID* and *SGID* are explained in Chapter 5.

The following in-chapter exercises familiarize you with the `echo`, `cd`, `ls`, and `file` commands and the formats of their output.

EXERCISE 4.3

Right after you log on, run `echo ~` to determine the full pathname of your home directory.

EXERCISE 4.4

Use the `cd` command to go to the **/usr/bin** directory on your system and run the `ls -F` command. Identify two symbolic links and five binary files.

EXERCISE 4.5

Run the `ls -l` command in the same directory and write down sizes (in bytes) of the `find` and `sort` commands.

EXERCISE 4.6

Run the `file /etc/*` command to identify types of all the files in this directory.

4.5.6.1 File Representation and Storage in UNIX

As stated earlier, the attributes of a file are stored in a data structure on the disk, called an inode. At the time of its creation, every file is allocated a unique inode from a list (array) of inodes on the disk, called the *i-list*. The index value of the inode in the i-list is called the inode number for the inode allocated to the file, and is known as the file's inode number.

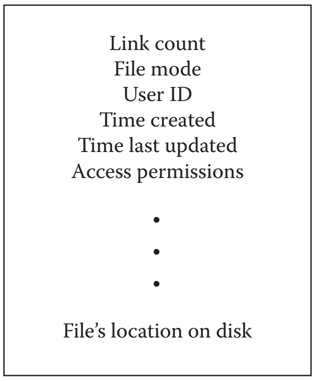


FIGURE 4.3 Contents of an inode.

The UNIX kernel also maintains a table of inodes, called the *inode table*, in the main memory for all open files. When an application opens a file, an inode is allocated from the inode table and the contents of the file's inode on the disk are copied into it. The inode number is used to index the inode table, allowing quick access to the attributes of an open file. When a file's attributes (e.g., file size) change, the inode in main memory is updated; disk copies of inodes are updated at fixed intervals. For files that are not open, their inodes reside on the disk. Some of the contents of an inode are shown in Figure 4.3.

The "link count" field specifies the number of different names the file has within the system. This count is also known as the *hard link count* (see Chapter 8 for details on links). The "file mode" field specifies what the file was opened for (read, write, etc.). The "user ID" is the ID of the owner of the file. The "access permissions" field specifies who can access the file for what type of operation (discussed in more detail in Chapter 5). The file's location on disk is specified by a number of *direct* and *indirect* pointers to disk blocks containing file data.

A typical computer system has several disk drives. Each drive consists of a number of platters with two *surfaces* (top and bottom). Each surface is logically divided into concentric circles called *tracks*, and each track is subdivided into fixed size portions called *sectors*. Tracks at the same position on both surfaces of all platters comprise a *cylinder*. Disk input/output (I/O) takes place in terms of one sector, also called a disk block. For this reason, disks are known as *block devices*. Traditionally, the sector size for hard disks has been 512 bytes. Newer hard disks use 4K-byte (i.e., 4096-byte) sector sizes. CD-ROMs and DVD-ROMs use 2K-byte (i.e., 2048-byte) sector sizes.

A sector may be addressed by using a four-dimensional address comprising <disk #, cylinder #, surface #, and sector #>. This four-dimensional address is translated to a *linear* (one-dimensional) block number, and most of the software in UNIX deals with block addresses because they are relatively easy to deal with. These blocks start with the sector numbered as 0 on the outermost cylinder on the topmost surface (i.e., the topmost track of the outermost cylinder), which is assigned block number 0. The block numbers increase through the rest of the tracks in that cylinder, through the rest of the cylinders on the disk, and then through the rest of the disks. The diagram shown in Figure 4.4 is a logical view of

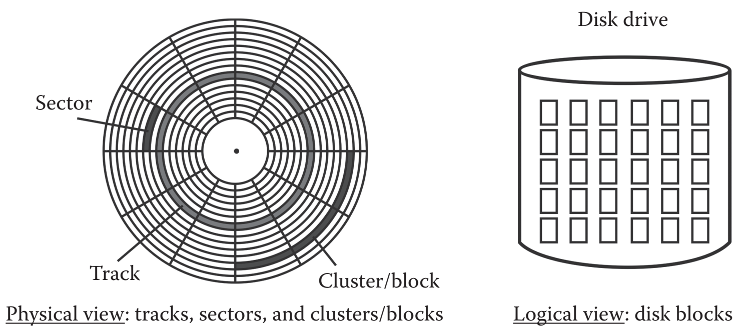

Sector
Track
Cluster/block
Physical view: tracks, sectors, and clusters/blocks
Disk drive
Logical view: disk blocks

FIGURE 4.4 Physical and logical views of a disk drive in terms of tracks, sectors, clusters, and disk blocks.

a disk system consisting of an array of disk blocks. File space is allocated in *clusters* of two, four, or eight disk blocks.

Figure 4.5 shows how an inode number for an open file can be used to access a file's attributes, including the file's contents, from the disk. It also shows contents of the directory **~/courses/ee446/labs** and how the UNIX kernel maps the inode of the file **lab1.c** to its contents on disk. As previously discussed, and as shown in the diagram, a directory consists of an array of entries <inode #, filename>. Accessing (reading or writing) the contents

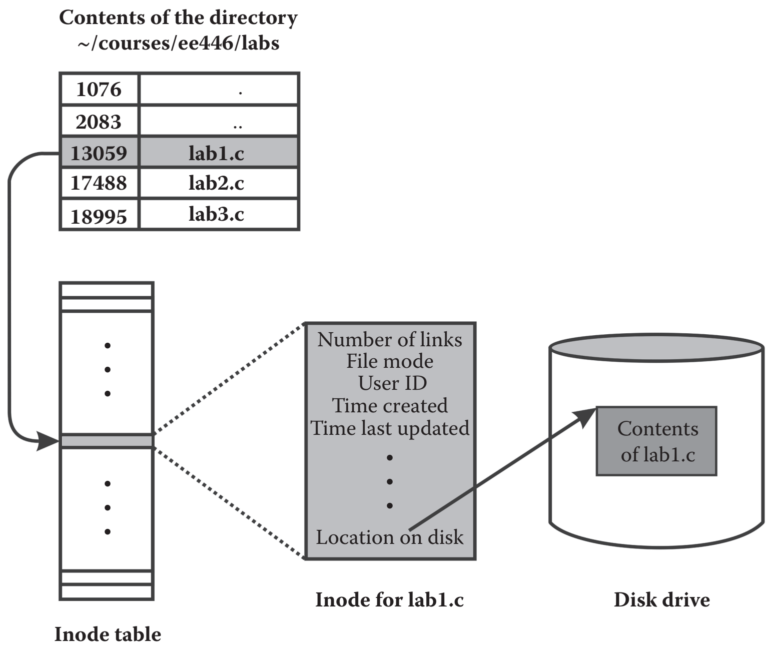


FIGURE 4.5 Relationship between the file **lab1.c** in a directory and its contents on a disk.

of **lab1.c** requires the use of its inode number to index the in-memory inode table to get to the file's inode. The inode, as previously stated, contains, among other things, the location of **lab1.c** on the disk.

The inode contains the location of **lab1.c** on the disk in terms of the numbers of the disk blocks that contain the contents of the file. The details of how exactly a UNIX file's location is specified in its inode and how it is stored on the disk are beyond the scope of this textbook. These details are available in any book on UNIX internals.

4.6 STANDARD FILES AND FILE DESCRIPTORS

When an application needs to perform an I/O operation on a file, it must first open the file and then issue the file operation (read, write, seek, etc.). UNIX automatically opens three files for every command it executes. The command reads input from one of these files and sends its output and error messages to the other two files. These files are called *standard files: standard input* (**stdin**) files, *standard output* (**stdout**) files, and *standard error* (**stderr**) files. By default, these files are attached to the terminal on which the command is executed. That is, the shell makes the command input come from the terminal keyboard, and its output and error messages go to the terminal (or the console window in case of an `ssh` session or an `xterm` in a UNIX system running the X Window System, as discussed in detail in Chapter 23). These default files can be changed to other files by using the redirection operators: < for input redirection and > for output and error redirection.

A small integer, called a *file descriptor*, is associated with every open file in UNIX. The integer values 0, 1, and 2 are the file descriptors for **stdin**, **stdout**, and **stderr**, respectively, and are also known as *standard file descriptors*. The kernel uses file descriptors to perform file operations (e.g., file read), as illustrated in Figure 4.6. The kernel uses a file descriptor to index the per-process file descriptor table to obtain a pointer to the system-wide file table. The file table, among other things, contains a pointer to the file's inode in the inode table. Once the inode for the file has been accessed, the required file operation is performed by

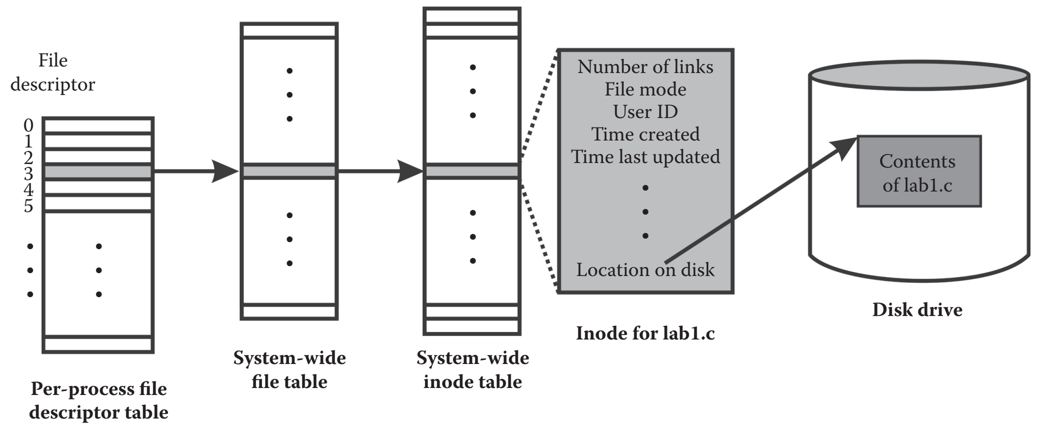


FIGURE 4.6 Relationship between file descriptors and the contents of files on disk.

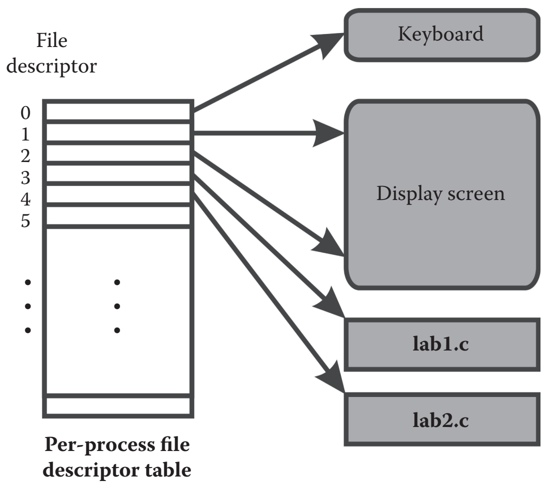


FIGURE 4.7 Logical view of the relationship between file descriptors and the corresponding files.

accessing appropriate disk block(s) for the file by using the direct and indirect pointers, as described in Section 4.6.

Recall that every device, including a terminal, is represented by a file in UNIX. The diagram shown in Figure 4.7 depicts the relationship between a file and its file descriptor. Here, we assume that files **lab1.c** and **lab2.c** are open for some file operations (say, file read) and have descriptors 3 and 4, respectively, that the kernel assigned to the files when they were opened. We have described the details of this relationship in the preceding paragraph and in Section 4.6, in terms of a file table, inode table, and the storage of the file on disk; also see Figures 4.5 and 4.6.

The UNIX system allows standard files to be changed to alternate files for a single execution of a command, including a shell script. This concept of changing standard files to alternate files is called *input*, *output*, and *error redirection*. We address input, output, and error redirection in detail in Chapter 9. We have briefly mentioned the standard files and file descriptors here because most UNIX commands that explicitly require input from an outside source get it from standard input, unless it comes from a file (or list of files) that is passed to the command as a command line argument. Similarly, most UNIX commands that produce output send it to standard output. This information is important for proper understanding and use of commands in the remaining chapters.

4.7 END-OF-FILE (eof) MARKER

Every UNIX file has an end-of-file (*eof*) marker. The commands that read their input from files read the eof marker when they reach the end of a file. For files that can be stored, the value of the eof marker is not a character; it is usually a small negative integer such as `-1`. The `<Ctrl-D>` on a new line is the UNIX eof marker when the input file is attached to a keyboard. That is why commands such as `cat` while reading input from the keyboard (see Chapter 6) terminate when you press `<Ctrl-D>` on a new line.

4.8 FILE SYSTEM

A file system is a directory hierarchy with its own root stored on a disk or disk partition, *mounted* under (glued to) a directory. The files and directories in the file system are accessed through the directory under which they are mounted. All references to a file system in this textbook are to the local file system. The description of the UNIX network file system (NFS) is beyond the scope of this textbook, but it is briefly mentioned in Chapter 11.

PC-BSD and Solaris use the ZFS file system, which was first released by Oracle for Solaris in June 2006. It became part of FreeBSD in April 2007. It is a POSIX-compliant file system that is reliable, flexible, and scalable, with built-in compression and advance features for file system backup and restoration. However, the hallmark of ZFS is its focus on maximum data integrity that protects user data against all sorts of errors, including those caused by decaying storage media, electric current spikes, accidental disk writes, and so on. It achieves such data integrity by using several techniques, including data replication. The ZFS file system's features and issues are discussed in more detail under "System Administration" in Chapter 23. Browse through the relevant websites listed in "Web Resources" (Table 4.4) to find out more about the ZFS file system.

SUMMARY

In UNIX, a file is a sequence of bytes. This simple, yet powerful, concept and its implementation lead to nearly everything in the system being a file; users and processes are not. UNIX supports seven types of file: ordinary file, directory, symbolic link, character special file, block special file, named pipe (also known as FIFO), and socket. No file extensions are supported for files of any type, but applications running on a UNIX system can require their own extensions.

Every file in UNIX has several attributes associated with it, including file name, owner's name, date last modified, link count, and the file's location on disk. These attributes are stored in an area on the disk called an inode. When files are opened, their inodes are copied to a kernel area called the inode table for faster access of their attributes. Every file in a directory has an entry associated with it that comprises the file's name and its inode

TABLE 4.4 Web Resources for UNIX File Systems

URL	Description
`http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/dirstructure.html`	This website contains a comprehensive description of the FreeBSD directory hierarchy.
`http://en.wikipedia.org/wiki/ZFS`	This website describes the ZFS file system.
`http://www.bsdnow.tv/tutorials/zfs` `http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/filesystems-zfs.html`	These websites contain very good tutorials on the ZFS file system, describing the installation of ZFS and administration of its various features.
`http://www.princeton.edu/~unix/Solaris/troubleshoot/zfs.html`	This website discusses the management of the ZFS file system.
`http://2007.asiabsdcon.org/papers/P16-paper.pdf`	This website contains a paper that describes porting ZFS to FreeBSD.

number. The kernel accesses an open file's attributes, including its contents, by reading the file's inode number from its directory entry and indexing the inode table with the inode number.

The UNIX file structure is hierarchical with a root directory and all the files and directories in the system under it. Every user has a directory called the user's home directory, which he or she is placed when logging on to the system. Multiple disk drives and/or disk partitions can be mounted on the same file system structure, allowing their access as directories, and not as named drives A:, B:, C:, and so on, as in MS-DOS and Microsoft Windows. This approach gives a unified view of all the files and directories in the system, and users do not have to worry about remembering the names of drives and the files (and directories) that they contain.

Directories (primarily) can be created and removed under the user's home directory. The file structure can be navigated by using various commands (`mkdir`, `rmdir`, `cd`, `ls`, etc.). You can specify a file in the system by using the file's absolute or relative pathname. An absolute pathname starts with the root directory, and a relative pathname starts with a user's home directory or present working directory.

UNIX automatically opens three files for every command for it to read input from and send its output and error messages to. These files are called standard input (**stdin**), standard output (**stdout**), and standard error (**stderr**). By default, these files are attached to the terminal on which the command is executed; that is, the command input comes from the terminal keyboard, and the command output and error messages go to the terminal's screen display. The default files can be changed to other files by using redirection primitives: < for input redirection, and > for output and error redirection.

The kernel associates a small integer with every open file. This integer is called the file descriptor. The kernel uses file descriptors to perform operations (e.g., read) on the file. The file descriptors for **stdin**, **stdout**, and **stderr** are 0, 1, and 2, respectively.

Every UNIX file has an end-of-file (eof) marker, which is a small negative integer such as `−1`. The eof marker is `<Ctrl-D>` if a command reads input from the keyboard.

QUESTIONS AND PROBLEMS

1. What is a file in UNIX?

2. Does UNIX support any file types? If so, name them. Does UNIX support file extensions?

3. What is a directory entry? What does it consists of?

4. What are special files in UNIX? What are character special and block special files? Run the `ls /dev | wc -w` command to find the number of special files your system has.

5. What is meant by interprocess communication? Name three tools that UNIX provides for interprocess communication.

6. Draw the hierarchical file structure, similar to the one shown in Figure 4.2, for your UNIX machine. Show files and directories at the first two levels. Also show where your home directory is, along with files and directories under your home directory.
7. Give three commands that you can use to list the absolute pathname of your home directory.
8. Write down the line in the **/etc/passwd** file on your system that contains information about your login. What are your login shell, user ID, home directory, and group ID? Does your system contain the encrypted password in the **/etc/passwd** or **/etc/shadow** file?
9. What would happen if the last field of the line in the **/etc/passwd** file were replaced with **/usr/bin/date**? Why?
10. What are the inode numbers of the root and your home directories on your machine? Give the commands that you used to find these inode numbers.
11. Create a directory in your home directory, called **memos**. Go into this directory and create a file called **memo.james** by using one of the editors such as vi. Give three pathnames for this file.
12. Give a command for creating a subdirectory called **personal** under the **memos** directory that you created in Problem 11.
13. Make a copy of the file **memo.james** and put it in your home directory. Name the copied file **temp.memo**. Give two commands for accomplishing this task.
14. Draw a diagram like that shown in Figure 4.5 for your **memos** directory. Clearly show all directory entries, including inode numbers for all the files and directories in it.
15. Give the command for deleting the **memos** directory. How do you know that the directory has been deleted?
16. Why does a shell process terminate when you press `<Ctrl-D>` at the beginning of a new line?
17. Give a command to display the types of all the files in your **~/unix** directory that start with the word chapter, are followed by one of the digits 1, 2, 6, 8, or 9, and end with **.eps** or **.prn**.
18. Give a command line to display the types of all the files in the personal directory in your home directory that do not start with letters a, k, G, or Q and the third letter in the name is not a digit and not a letter (uppercase or lowercase).
19. Use the `ls -i` command to display inode numbers for the /, **/usr**, and ~ directories on your system. Show outputs of your commands and identify the inode numbers for these directories.

20. Display the absolute pathnames of your home directory by using two different methods in the Bourne and C shells. Does your system use any symbolic links in the / directory? If so, display those symbolic links by using a shell command.

21. What are the basic characteristics of the ZFS file system? When did it first become part of Solaris and which version? When did FreeBSD adopt it?

22. What is the maximum size (in bytes/characters) of the file name in your UNIX system? What command did you use to obtain your answer?

23. Browse through the following directories and identify five commonly used commands, tools, utilities, and daemons in them: **/bin**, **/sbin**, **/usr/bin**, **/usr/sbin**, **/usr/local/bin**, and **/usr/local/sbin**. Clearly state the purpose of each command, tool, utility, or daemon and the name of the directory in which it is found.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

CHAPTER 5

File Security

Objectives

- To show the three protection and security mechanisms that UNIX provides
- To describe the types of users of a UNIX file
- To discuss the basic operations that can be performed on a UNIX file
- To explain the concept of file access permissions/privileges in UNIX
- To discuss how a user can determine access privileges for a file
- To describe how a user can set and change permissions for a file
- To cover the commands and primitives

```
?, ~, *, chmod, groups, ls -l, ls -ld, umask, umask -S
```

5.1 INTRODUCTION

As we pointed out earlier, a time-sharing system offers great benefits. However, it poses the main challenge of protecting the hardware and software resources in it. These resources include the input/output (I/O) devices, central processing unit (CPU), main memory, and the secondary storage devices that store user files. The CPU runs user and kernel processes, the main memory stores user processes and important operating system code and data structures while the system is running, and the secondary storage devices store user files and operating system code on a permanent basis. We limit this chapter to a discussion about the protection of a user's files from unauthorized access by other users. UNIX provides three mechanisms to protect your files.

The most fundamental scheme to protect user files is to give every user a login name and a password, allowing a user to use a system (see Chapter 2). To prevent others from accessing your files, keep the password for your computer account strictly confidential. The second scheme protects individual files by converting them to a form that is completely different from the original version by means of encryption. This technique is used to protect your most important files, so that the contents of these files cannot be understood even if someone somehow gains access to them on the system. The third file protection scheme allows you to protect your files by associating access privileges with them, so that only a subset of users can access these files for a subset of file operations. In other words, the owner of the files can decide to whom to grant access to these files. All three mechanisms are described in this chapter, with emphasis on the third scheme.

5.2 PASSWORD-BASED PROTECTION

The first mechanism that allows you to protect your files from other users is the login password scheme. Every user of a UNIX-based computer system is assigned a login name (a name by which the user is known to the UNIX system) and a password. Both the login name and password are assigned by the system administrator and are required for a user to enter and use a UNIX system. All login names are public knowledge and can be found in the **/etc/passwd** file. A user's password, however, is given to that user only. This scheme prevents users from accessing each other's files. Users are encouraged to change their passwords frequently by using the `passwd` command (see Chapter 2). On some networked systems, you may have to use the `yppasswd` or `nispasswd` command to change your password on all the network's computer systems. PC-BSD has the `yppasswd` command and you can browse through its manual page to learn about it. Consult your instructor about the command that you need to use on your particular system. Usually, only the system administrator can change login names. Under some UNIX installations, users are also allowed to change their usernames.

The effectiveness of this protection scheme depends on how well protected a user's password is. If someone knows your password, that person can log on to the system and access your files. There are, primarily, three ways of discovering a user's password.

1. You, as the owner of an account, inform others of your password.
2. A user's password can be guessed by another user.
3. A user's password can be guessed by "brute force."

Never let anyone else know your password under any circumstances. As a safety measure, you should change your password regularly. Always choose passwords that would be difficult for others to guess. A good password is one that is a mixture of letters, digits, and punctuation marks—but it must be easy for you to memorize. Never write your password on a piece of paper, and never use birthdays or the names of relatives, friends, famous sportspersons, or favorite movie actors as passwords. Also, avoid using words as passwords.

By using "brute force," someone tries to learn your password by trying all possible combinations of characters until the user's password is found. Guessing someone's password is a time-consuming process and is commonly used by hackers. The brute force method can be made more time-consuming for an infiltrator if the password is long and consists of letters, digits, and punctuation marks. To illustrate the significance of using a more complex password, consider a system in which the password is exactly eight characters consisting of decimal digits only. This would allow a maximum of 10^8 (100 million) passwords that the brute force method would have to go through, in the worst-case analysis. If the same system requires passwords to consist of a mixture of digits and uppercase letters (a total of 36 symbols: 10 digits and 26 uppercase letters), the password space would comprise 36^8 (about 2.8 trillion) passwords. If the system requires passwords that consist of a mixture of digits, uppercase letters, and lowercase letters, the password space would comprise 62^8 (about 218 trillion) passwords. Imagine the size of the password space if passwords could include punctuation marks too! Many systems force a short (e.g., 5 s) delay after an invalid password is entered before the next login prompt, to make the infiltrator's job even harder.

The following in-chapter exercise asks you to figure out how to change your password on your system.

EXERCISE 5.1

In some UNIX systems you are not allowed to change your password. Does your system allow you to change your password? If so, change your password. What command did you use?

Note: Be sure to memorize your new password, because if you forget it you will have to request your system administrator to reset your password to a new value, unless your system allows you to change your password back to the previous password.

5.3 ENCRYPTION-BASED PROTECTION

In the second protection scheme, a software tool is used to convert a file to a form that is completely different from its original version. The transformed file is called an *encrypted file*, and the process of converting a file to an encrypted file is called *encryption*. The same tool is used to perform the reverse process of transforming the encrypted file to its original form, called *decryption*. You can use this technique to protect your most important files so that their contents cannot be understood even if someone else gains access to them. Figure 5.1 illustrates the encryption and decryption processes.

The UNIX command `crypt` can be used to encrypt and decrypt your files. You can learn more about this command by running the `man crypt` command. This command is discussed in detail in Chapter 7.

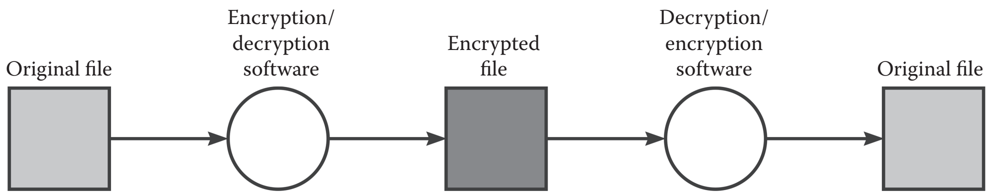

Original file
Encryption/ decryption software
Encrypted file
Decryption/ encryption software
Original file

FIGURE 5.1 Process of encryption and decryption.

5.4 PROTECTION BASED ON ACCESS PERMISSION

The third type of file protection mechanism prevents users from accessing each other's files when they are not logged on as a file's owner. As file owner, you can attach certain access rights to your files that dictate who can and cannot access them for various types of file operations. This scheme is based on the types of users, the types of access permissions, and the types of operations allowed on a file under UNIX. Without this protection scheme, users can access each other's files because the UNIX file system structure (see Figure 4.2) has a single root from which all the files in the system hang. We use the terms access permissions, access privileges, and access rights synonymously throughout the book.

5.4.1 Types of Users

Each user in a UNIX system belongs to a group of users, as assigned by the system administrator when a user is allocated an account on the system. A user can belong to multiple groups, but a typical UNIX user belongs to a single group. All the groups in the system and their memberships are listed in the file **/etc/group** (see Figure 4.2). This file contains one line per group, with the last field of the line containing the login names of the group members. A user of a file can be the owner of the file, a user who belongs to the same group as the owner, or everyone else who has an account on the system. These, respectively, comprise the three types of users of a UNIX file: *user*, *group*, and *others*. As the owner of a file, you can specify who can access it. The group name of a file is known as the *group owner* of the file.

Once a user of a system has logged in, he/she is known to the UNIX system by an integer number, known as the *user ID* (UID), and not by the user's login name. Every UNIX system has one special user who has access to all of the files on the system, regardless of the access privileges on the files. This user manages (administers) your UNIX system and is commonly known as the *superuser* or *system administrator*. The login name for the superuser is **root**, and the user ID is **0**.

You can see the list of all user groups on your system by displaying the **/etc/group** file, shown as follows.

```
$ more /etc/group
root::0:root,davis
other::1:
bin::2:root,bin,daemon
sys::3:root,bin,sys,adm
adm::4:root,daemon,adm
uucp::5:root,uucp
mail::6:root
tty::7:root,tty,adm
lp::8:root,lp,adm
nuucp::9:root,nuucp
staff::10:
daemon::12:root,daemon
sysadmin::14:davis
```

```
nobody::60001:
noaccess::60002:
nogroup::65534:
utadmin::100:
faculty::101:
ra::102:
courses::103:zartash
student::104:ank
...
$
```

There is one line in this file for every group on the system, each line having four colon-separated fields. The first field specifies the group name, the second specifies some information about the group, the third specifies the group ID as a number, and the last specifies a comma-separated list of users who are members of the group. For example, the **bin** group has group ID **2** and its members are users **root**, **bin**, and **daemon**. If the membership list in a line is missing (e.g., the **faculty** group), this means that its membership is specified in the **/etc/passwd** file. The system administrator makes a user part of a group at the time of adding the user to the system. This group is known as a user's *default group*. The default group membership of a user is specified in the user's entry in the **/etc/passwd** file. The system administrator can make a user part of another group (in addition to his/her default group) by placing his/her username in the comma-separated list of members for the group. You can use the `groups` command to display which groups on your system a user is a member of. The following session shows that **sarwar** is a member of the **faculty** group only; **zartash** belongs to the groups **faculty** and **courses**; **davis** is a member of three groups: **faculty**, **root**, and **sysadmin**; and **root** is a member of 11 groups: **other**, **root**, **bin**, **sys**, **adm**, **uucp**, **mail**, **tty**, **lp**, **nuucp**, and **daemon**.

```
$ groups sarwar
faculty
$ groups zartash
faculty courses
$ groups davis
faculty root sysadmin
$ groups root
other root bin sys adm uucp mail tty lp nuucp daemon
$
```

5.4.2 Types of File Operations/Access Permissions

In UNIX, three types of access permissions/privileges can be associated with a file: read (r), write (w), and execute (x). The read permission on a file allows you to read the file, the write permission allows you to write to or remove the file, and the execute permission allows you to execute/run the file. The execute permission should be set for executable files only, i.e., files containing binary code (i.e., executable code generated by a compiler) or

TABLE 5.1 Summary of File Permissions in UNIX

	Permission Type		
User Type	**Read (r)**	**Write (w)**	**Execute (x)**
User (u)	X	X	X
Group (g)	X	X	X
Others (o)	X	X	X

shell scripts, as setting it for any other type of file does not make any sense. We discuss the purpose of execute permission on a directory in Section 5.4.3.

With three categories of file users and three types of permissions for each user type, a UNIX file has nine types of permissions associated with it, as shown in Table 5.1. Note that permissions are read across row by row. As stated in Chapter 4, access privileges are stored in a file's inode.

The value of X can be 1 (permission granted) or 0 (permission not granted). Thus, one bit is needed to represent a permission type and a total of three bits are needed to indicate file permissions for one type of user. In other words, a user of a file can have one of the eight (2^3) possible types of permissions for a file at a given time. Octal numbers can represent these eight three-bit access permission values from 0 to 7, as shown in Table 5.2. Access permissions 0 (binary 000) and 7 (binary 111) mean no access permissions and all permissions, respectively.

The nine bits needed to express permissions for the three types of file users result in the possible access permission values of three-digit octal numbers 000–777 for file permissions. The first octal digit specifies permissions for the owner of the file, the second digit specifies permissions for the group that the owner of the file belongs to, and the third digit specifies permissions for everyone else. In the output of the `ls -l` command, a bit value of 0 for a permission is displayed as a dash (`-`), and a value of 1 is displayed as `r`, `w`, or `x`, depending on the position of the bit according to the table. Thus a permission value of 0 in octal (no permissions granted) for a user of a file can be written as `---` and a permission of 7 (all three permissions granted) can be denoted as `rwx`. The outputs of the `ls -l` commands in the following session show that the **/etc/passwd** file is read-only for everyone on the system

TABLE 5.2 Possible Access Permission Values for a File for a User, Their Octal Equivalents, and Their Meanings

r	**w**	**x**	**Octal Digit for Permission**	**Meaning**
0	**0**	**0**	**0**	No permission
0	**0**	**1**	**1**	Execute-only permission
0	**1**	**0**	**2**	Write-only permission
0	**1**	**1**	**3**	Write and execute permissions
1	**0**	**0**	**4**	Read-only permission
1	**0**	**1**	**5**	Read and execute permissions
1	**1**	**0**	**6**	Read and write permissions
1	**1**	**1**	**7**	Read, write, and execute permissions

except root, who has read and write permissions. The **client.c** file has read and write permissions for the owner (**sarwar**) and group (**faculty**), and read-only permission for others.

```
$ ls -l /etc/passwd
-rw-r--r-- 1 root sys 33020 Jul 10 15:47 /etc/passwd
$ ls -l client.c
-rw-rw-r-- 1 sarwar faculty 1277 Dec 19 07:30 client.c
$
```

Because a user can be in many groups, you can expand access to your files to accommodate different users through groups.

5.4.3 Access Permissions for Directories

Next, we will look at what the read, write, and execute permissions mean for directories. The read permission for a directory allows you to read the contents of the directory; recall that the contents of a directory are the names of files and directories in it. Thus, the `ls` command can be used to list its contents. The write permission for a directory allows you to create a new directory or a file in it or to remove an existing entry from it. The execute permission for a directory is permission to search the directory but not to read from or write to it. Thus, if you do not have execute permission for a directory, you cannot use the `ls -l` command to list its contents or use the `cd` command to make it your current directory. The same is true if any component in a directory's pathname does not contain execute permission. We demonstrate these aspects of the search permission on directories in Section 5.5.2.

5.5 DETERMINING AND CHANGING FILE ACCESS PRIVILEGES

The following sections describe how you can determine the access privileges for files and directories and how you can change them to enhance or limit someone's access to your files.

5.5.1 Determining File Access Privileges

You can use the `ls` command with the `-l` or `-ld` option to display access permissions for a list of files and/or directories. The following is a brief description of the `ls` command with the two options.

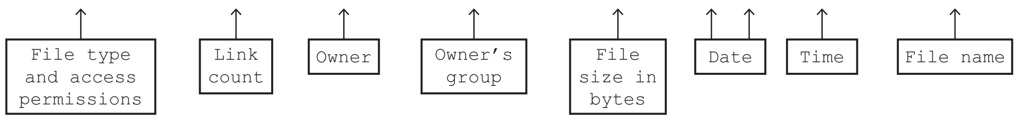
SYNTAX

```
ls -l [file-list]
ls -ld [directory-list]
```

Purpose: First syntax: Display the long list of files and/or directories in the space-separated **file-list** on the display screen; in the case where **file-list** contains directories, display long list of all the files in these directories
Second syntax: Display the long list of directories in **directory-list** on the display screen.

If no **file-list** is specified, the command gives long lists for all the files (except hidden files) in the present working directory. Add the `-a` option to the command line to include the hidden files in the display. Consider the following session.

```
$ ls -l
drwxr-x---     2   sarwar  faculty    512   Jul 29   17:35   courses
-rwxrwxrwx     1   sarwar  faculty     12   May 01   13:22   labs
-rwxr--r--     1   sarwar  faculty    163   May 05   23:13   temp
$
```



The leftmost character in the first field of the output indicates the file type (`d` for a directory and — for an ordinary file). The remaining nine characters in the first field show the file access privileges for user, group, and others, respectively. The second field indicates the number of hard links (discussed in Chapter 8) to the file. The third field shows the owner's login name. The fourth field shows the file's group owner. The fifth field displays the file's size (in bytes). The sixth, seventh, and eighth fields display the date and time of file's creation (or last update). The last field is the file's name. Table 5.3 shows who has what type of access privileges for the three files in this session: **courses**, **labs**, and **temp**.

If an argument of the `ls -l` command is a directory, the command displays the long lists of all the files and directories in it. You can use the `ls -ld` command to display the long lists of directories only. When executed without an argument, this command displays the long list for the current directory, as shown in the first command of the following session. The second and third commands show that when the `ls -ld` command is executed with a list of directories as its arguments, it displays the long lists for those directories only. If an argument to the `ls -ld` command is a file, the command displays the long list for the file. The fourth command, `ls -ld pvm/*`, displays the long lists for all the files and directories in the **pvm** directory.

```
 $ ls -ld
 drwx--x--x 2  sarwar  faculty  11264 Jul   8   22:21 .
 $ ls -ld ABET
 drwx------ 2  sarwar  faculty    512 Dec   18  1997 ABET
$ ls -ld ~/Images courses/ee446
drwx------  3  sarwar  faculty    512 Apr   30  09:52  courses/ee446
```

TABLE 5.3 Permissions for Access to the Files **courses**, **labs**, and **temp** for the Three Types of Users

File Name	User	Group	Other
courses	Read, write, search	Read and search	No permission
labs	Read, write, execute	Read, write, execute	Read, write, execute
temp	Read, write, execute	Read	Read

```
drwx--x--x  2  sarwar  faculty   2048 Dec   18  1997   /home/sarwar/
Images
$ ls -ld pvm/*
drwx------  3  sarwar  faculty    512 Dec   18  1997   pvm/examples
drwx------  2  sarwar  faculty   1024 Oct   27  1998   pvm/qsort
-rw-------  1  sarwar  faculty   1606 Jun   19  1995   pvm/Book_PVM
-rw-------  1  sarwar  faculty   7639 Sep   11  1998   pvm/Jim_Davis
$
```

5.5.2 Changing File Access Privileges

You can use the chmod command to change access privileges for your files. The following is a brief description of the command.

SYNTAX

```
chmod [options] octal-mode file-list
chmod [options] symbolic-mode file-list
```

Purpose: Change/set permissions for files in **file-list**

Commonly used options/features:

- -R Recursively descend through directories changing/setting permissions for all the files and subdirectories under each directory
- -f Force specified access permissions; no error messages are produced if you are the owner of the file

The *symbolic mode*, also known as *mode control word*, has the form <who><operator><privilege>, with possible values for "who," "operator," and "privilege" shown in Table 5.4. This table also shows the use of + and – operators in the chmod command to add and remove a permission.

Note that u, g, or o can be used as a privilege with the = operator only. Multiple values can be used for "who" and "privilege," such as ug for the "who" field and rx for the

TABLE 5.4 Values for Symbolic Mode Components

Who	Operator	Privilege
u User	+ Add privilege	r Read bit
g Group	– Remove privilege	w Write bit
o Other	= Set privilege	x Execute/search bit
a All		u User's current privileges
ugo All		g Group's current privileges
		o Others' current privileges
		l Locking privilege bit
		s Sets user or group ID mode bit
		t Sticky bit

TABLE 5.5 Examples of the chmod Command and Their Purposes

Command	Purpose
`chmod 700 *`	Sets access privileges for all files, including directories, in the current directory to read, write, and execute for the owner, and provides no access privilege to anyone else
`chmod 740 courses`	Sets access privileges for courses to read, write, and execute for the owner and read-only for the group, and provides no access for others
`chmod 751 ~/courses`	Sets access privileges for **~/courses** to read, write, and execute for the owner, read and execute for the group, and execute-only permission for others
`chmod 700 ~`	Sets access privileges for the home directory to read, write, and execute (i.e., search) for the owner, and no privileges for anyone else
`chmod u=rwx courses`	Sets owner's access privileges for **courses** to read, write, and execute and keeps the privileges of group and others at their present values
`chmod ugo-rw sample` or `chmod a-rw sample`	Does not let anyone read or write **sample**
`chmod a+x sample`	Gives everyone execute permission for **sample**
`chmod g=u sample`	Makes **sample**'s group privileges match its user (owner) privileges
`chmod go= sample`	Removes all access privileges to **sample** for group and others

"privilege" field. Some useful examples of the chmod command and their purposes are listed in Table 5.5.

The following session illustrates how access privileges for files can be determined and set. The chmod command is used to change (or set) access privileges, and the ls -l (or ls -ld) command is used to show the effect of the corresponding chmod command. After the chmod 700 courses command has been executed, the owner of the **courses** file has read, write, and execute access privileges for it, and nobody else has any privilege. The chmod g+rx courses command adds read and execute access privileges to the **courses** file for the group; the privileges of the owner and others remain intact. The chmod o+r courses command adds the read access privilege for the **courses** file for others. The chmod a-w * command takes away the write access privilege from all users for all the files in the current directory. The chmod 700 [l-t]* command sets the access permissions to 700 for all the files that start with letters "l" through "t," as illustrated by the output of the last ls -l command, which shows access privileges for the files **labs** and **temp** changed to 700.

```
$ cd
$ ls -l
drwxr-x---   2   sarwar  faculty   512  Apr   23   09:37   courses
-rwxrwxrwx   1   sarwar  faculty    12  May   01   13:22   labs
-rwxr--r--   1   sarwar  faculty   163  May   05   23:13   temp
$ chmod 700 courses
$ ls -ld courses
drwx------   2   sarwar  faculty   512  Apr   23   09:37   courses
```

```
$ chmod g+rx courses
$ ls -ld courses
drwxr-x---   2   sarwar  faculty   512  Apr   23   09:37   courses
$ chmod o+r courses
$ ls -ld courses
drwxr-xr--   2   sarwar  faculty   512  Apr   23   09:37   courses
$ chmod a-w *
$ ls -l
dr-xr-x---   2   sarwar  faculty   512  Apr   23   09:37   courses
-r-xr-xr-x   1   sarwar  faculty    12  May   01   13:22   labs
-r-xr—r---   1   sarwar  faculty   163  May   05   23:13   temp
$ chmod 700 [l-t]*
$ ls -l
dr-xr-x---   2   sarwar  faculty   512  Apr   23   09:37   courses
-rwx------   1   sarwar  faculty    12  May   01   13:22   labs
-rwx------   1   sarwar  faculty   163  May   05   23:13   temp
$
```

The access permissions for all the files and directories under one or more directories can be set by using the `chmod` command with the `-R` option. In the following session, the first command sets access permissions for all the files and directories under the directory called **courses** to 711, recursively. The second command sets access permissions for all the files and directories under **~/personal/letters** to 700, recursively. “Recursively” means by traversing all subdirectories under the specified directories, i.e., **courses** and **~/personal/letters** in these examples.

```
$ chmod -R 711 courses
$ chmod -R 700 ~/personal/letters
$
```

If you specify access privileges with a single octal digit in a `chmod` command, it is used by the command to set the access privileges for “others”; the access privileges for “user” and “group” are both set to 0 (i.e., no access privileges). If you specify two octal digits in a `chmod` command, the command uses them to set access privileges for “group” and “others”; the access privileges for “user” are set to 0. In the following session, the first `chmod` command sets “others” access privileges for the **courses** directory to 7 (`rwx`) and 0 (`---`) for owner and group. The second `chmod` command sets “group” and “others” access privileges for the **personal** directory to 7 (`rwx`) and 0 `(---)`, respectively, and no access rights for the file owner. The `ls -l` command shows the results of these commands.

```
$ chmod 7 courses
$ chmod 70 personal
$ ls –l
d------rwx  2  sarwar faculty  512   Nov  10   09:43   courses
d---rwx---  2  sarwar faculty  512   Nov  10   09:43   personal
drw-------  2  sarwar faculty  512   Nov  10   09:43   sample
$
```

5.5.3 Access Privileges for Directories

As previously stated, the read permission on a directory allows you to read the contents of the directory (recall that the contents of a directory are the names of files and directories in it), the write permission allows you to create a file in the directory or remove an existing file or directory from it, and the execute permission for a directory is permission for searching the directory. It is important to note that read and write permissions on directories are not meaningful without the search permission. So, you must have both read and execute permissions on a directory to be able to list its contents. Similarly, you must have both write and execute permissions on a directory to be able to create a file in it.

In the following session, the write permission for the directory **courses** has been turned off. Thus, you cannot create a subdirectory **ee345** in this directory by using the `mkdir` command or copy a file **foo** into it. Similarly, as you do not have search permission for the directory **personal**, you cannot use the `cd` command to enter (change directory to) this directory. If the directory **sample** had a subdirectory, say **foobar**, for which the execute permission was turned on, you still could not change directory to **foobar**, because search permission for **sample** is turned off. Finally, as read permission for the directory **personal** is turned off, you cannot display the names of files and directories in it by using the `ls` command, even though search permission on it is turned on.

```
$ chmod 600 sample
$ chmod 500 courses
$ chmod 300 personal
$ ls -ld courses personal sample
dr-x------ 2 sarwar sarwar 512 Aug 4 06:36 courses
d-wx------ 2 sarwar sarwar  62 Aug 4 06:36 personal
drw------- 2 sarwar sarwar  88 Aug 4 06:36 sample
$ mkdir courses/ee345
mkdir: courses/ee345: Permission denied
$ cp foo courses
cp: courses/foo: Permission denied
$ cd sample
cd: sample: Permission denied
$ ls -l personal
total 0
ls: personal: Permission denied
$
```

The next session shows that simply having read or write permission on a directory is not sufficient to read its contents (e.g., display them with the `ls` command) or create a file or directory in it. For example, the directory **dir1** has write permission turned on, but you cannot copy the **prog1.cpp** file into it, because search permission on it is turned off. Similarly, you cannot remove the file **f1** from **dir2**. After you turn on its search permission with the `chmod u+x dir2` command, you can remove the file **f1**.

```
$ ls -ld dir?
d-w------- 2 sarwar sarwar 2 Aug 4 06:59 dir1
d-w------- 2 sarwar sarwar 3 Aug 4 06:59 dir2
$ cp prog1.cpp dir1
cp: dir1/prog1.cpp: Permission denied
$ rm dir2/f1
rm: dir2/f1: Permission denied
$ chmod u+x dir2
$ ls -ld dir2
d-wx------ 2 sarwar sarwar 3 Aug 4 06:59 dir2
$ rm dir2/f1
$
```

The following in-chapter exercises ask you to use the `chmod` and `ls -ld` commands to see how they work, and to enhance your understanding of UNIX file access privileges.

EXERCISE 5.2

Create three directories called **courses**, **sample**, and **personal** by using the `mkdir` command. Set access permissions for the directory **sample** so that you have all three privileges, users in your group have read access only, and the other users of your system have no access privileges. Show your work.

EXERCISE 5.3

Use the `chmod o+r sample` command to allow others read access to the directory **sample**. Use the `ls -ld sample` command to confirm that read permission for **sample** has been enabled for others.

EXERCISE 5.4

Use the session preceding these exercises to understand fully how the read, write, and execute permissions work for directories. Run the session on your system and verify results.

5.5.4 Default File Access Privileges

When a new file or directory is created, UNIX sets its access privileges based on the current *mask* value. On UNIX systems, the default access privileges for the newly created files and directories are 777 for executable files and directories and 666 for text files. However, these default permissions may be different depending on the value of the mask set on your system. If a mask bit is 1, the corresponding permission will be disabled, and if it is 0, the permission will be determined by the system using a Boolean logic expression involving the current value of the mask and the default permissions. We describe this expression later in this section.

You can display the current mask value or set it to a new value by using the `umask` command. The following is a brief description of the command.

SYNTAX

```
umask [-S] [mask]
```

Purpose: Set access permission bits for newly created files and directories to 0, if the corresponding bit in the mask is set to 1. Other permission bits are determined by using a Boolean logic expression involving the values of the mask and default permissions. Without any argument, the command displays the current value of the mask. With the –S option, the command displays the mask value in a symbolic form (see the following paragraphs).

When the command is executed without an argument, it displays the current value of the bit mask in octal, as in shown in the first command of the following session. The rightmost nine bits (i.e., the rightmost three octal digits) are for user, group, and others, and the leftmost three bits (i.e., the leftmost octal digit) are for special access bits described in Section 5.6 below. The `umask -S` command displays the symbolic value of the mask, showing the access privileges that will be set for a newly created directory or executable file for user, group, and others.

```
$ umask
0022
$ umask -S
u=rwx,g=rx,o=rx
$
```

When used with a mask as an argument, the `umask` command can be used to set the mask value. The mask value may be specified as a three-digit or four-digit octal number. In the following session, we show how the `umask` command can be used without and with the `–S` option.

```
$ umask 077
$ umask
0077
$ umask -S u=rwx,g=,o=
$ umask
0077
$
```

The `umask` command is normally placed in the system startup file **~/.profile** in System V UNIX and the **~/.login** or **~/.cshrc** file in BSD UNIX, so that it executes every time you log on to the system.

The argument of `umask` is a bit mask, specified in octal, that identifies the permission bits that are to be turned *off* when a new file is created. The values of other access permission bits are computed by the Boolean expression

```
A = B AND C' = BC'
```

Here, `A` is the file access permissions assigned to a newly created file or directory, `B` is the default access permission (777 for a directory or executable file and 666 for a text file), and `C` is the current mask value. `C′` (pronounced "NOT C" or "C prime") is called *negation*, or 1's complement of `C`. The 1's complement of a binary number is obtained by replacing 1s with 0s and vice versa. For example, the 1's complement of the four-bit binary number 1011 is 0100. The bitwise Boolean function `AND` of two binary variables returns 1 if and only if both the bits are 1; it returns 0 otherwise. While computing `A` in the above equation, the `AND` function compares the respective bits in `B` and `C′` and returns a 1 if and only if both bits are 1; otherwise it returns 0.

We now show a few examples how UNIX assigns file access permissions to newly created files and directories for a given mask. We determine the access permissions for a newly created directory or executable file for the mask value 022 by using the above Boolean expression.

```
C  = 022       = 000 010 010
C'             = 111 101 101
B  = 777       = 111 111 111
A  = B AND C' = 111 101 101 = 755 (octal) = 111101101 (binary)
                                            rwxr-xr-x (symbolic)
```

Thus, file access permissions are 755 (read, write, execute for user, read and execute for group and others). For a text file, B is 666. Thus, access permissions for a newly created text file would be 644, as follows:

```
C'             = 111 101 101
B = 666        = 110 110 110
A = B AND C'   = 110 100 100 = 644 (octal) = 110100100 (binary)
                                             rw-r--r-- (symbolic)
```

The access permissions for a newly created text file for the mask value 077 would be 600, as follows:

```
C = 077        = 000 111 111
C'             = 111 000 000
B = 666        = 110 110 110
A = B AND C'   = 110 000 000 = 600 (octal) = 110000000 (binary)
                                             rw------- (symbolic)
```

We now do a Bourne shell session to verify the above results. Note that with the mask set to 022, the permissions for the newly created directory (**labs**) and executable file (**hello**) are 755 (`rwx1-xr-x`), and are 644 (`rw-r--r--`) for the newly created text file, **tempfile**, as calculated above. With the mask set to 077, the permissions for the directory (**lectures**) and executable file (**greeting**) are 700 (`rwx------`), and are 600 (`rw-------`) for the text file (**textfile**).

```
$ umask 022
$ mkdir labs
$ cc hello.c -o hello
$ cat > tempfile
date
pwd
$ ls -ld labs hello tempfile
-rwxr-xr-x 1 sarwar sarwar 6945 Aug 5 22:27 hello
drwxr-xr-x 2 sarwar sarwar    2 Aug 5 22:26 labs
-rw-r--r-- 1 sarwar sarwar    9 Aug 5 22:30 tempfile
$ umask 077
$ mkdir lectures
$ cc hello.c -o greeting
$ cat > testfile
date
echo "Hello, world!"
$ ls -ld lectures greeting testfile
-rwx------ 1 sarwar sarwar 6945 Aug 5 22:28 greeting
drwx------ 2 sarwar sarwar    2 Aug 5 22:27 lectures
-rw------- 1 sarwar sarwar   26 Aug 5 22:29 testfile
$
```

The authors prefer a mask value of 077 so that their new files are always created with full protection in place, that is, files have full access permissions for the owner and no permissions for anyone else. This mask allows you to have a completely private system, not allowing other users to read, write, or execute your files or read, write, or search your directories. Recall that you can change access privileges for files on an as-needed basis by using the `chmod` command. Another common `umask` value is 027, which gives default privileges to group members and no permissions to others.

The following in-chapter exercise asks you to use the `umask` command to determine the current file protection mask.

EXERCISE 5.5

Run all the shell sessions shown and discussed in this section to make sure that they work on your system too and to understand the use of the `umask` command works and how the UNIX system decides about the access permissions on newly created files and directories.

5.6 SPECIAL ACCESS BITS

In addition to the nine commonly used access permission bits described in this chapter, three additional bits are of special significance. These bits are known as the set-user-ID (SUID) bit, set-group-ID (SGID) bit, and sticky bit.

5.6.1 Set-User-ID (SUID) Bit

We have previously shown that the external shell commands have corresponding files that contain binary executable codes or shell scripts. The programs contained in these files are not special in any way in terms of their ability to perform their tasks. Normally, when a command executes, it does so under the access privileges of the user who issues the command, which is how the access privileges system described in this chapter works. However, a number of UNIX commands need to write to files that are otherwise protected from users who normally run these commands. An example of this file is **/etc/passwd**, the file that contains a user's login information (see Chapter 4). Only the superuser is allowed to write to this file to perform tasks such as adding a new login and changing a user's group ID. However, UNIX users are normally allowed to execute the `passwd` command to change their passwords. Thus, when a user executes the `passwd` command, the command changes the user password in the **/etc/passwd** file on behalf of the user who runs this command. The problem is that we want users to be able to change their passwords, but at the same time they must not have write access to the **/etc/passwd** file to keep information about other users in this file from being compromised. Note that, on Solaris, it is the non-readable master password file **/etc/shadow** that changes.

As previously stated, when a command executes, it runs with the privileges of the user running the command. Another way of stating the same thing is that, when a command runs, it executes with the *effective user ID* of the user running the command. UNIX has an elegant mechanism that solves the problem stated in the preceding paragraph—and many other similar security problems—by allowing commands to change their effective user ID and become privileged in some way. This mechanism allows commands such as `passwd` to perform their work, yet not compromise the integrity of the system. Every UNIX file has an additional protection bit, called the SUID bit, associated with it. If this bit is set for a file containing an executable program, the program takes on the privileges of the owner of the file when it executes. Thus, if a file is owned by **root** and has its SUID bit set, it runs with superuser privileges. This bit is set, for example, for the `passwd` command. So, when you run the `passwd` command, it can write to the **/etc/passwd** file (replacing your existing password with the new password), even though you do not have access privileges to write to the file.

Several other UNIX commands require **root** ownership and the SUID bit set because they access and update operating system resources (files, data structures, etc.) that an average user must not have permissions for. Some of these commands are `lp`, `mail`, `mkdir`, `mv`, and `ps`. The authors of computer game software that maintains a scores file can make another use of the SUID bit. When the SUID bit is set for such software, it can update the scores file when a user plays the game, although the same user cannot update the scores file by explicitly writing to it.

The SUID bit is enabled if the execute bit for the owner is **s** (or **S**). If both SUID and execute bits are enabled, the bit is displayed as `s` in the output of the `ls –l` command. If SUID bit is enabled but execute bit is disabled, the bit is displayed as `S` by the `ls –l` command. The following session shows an example of each case.

```
$ ls -l cp.new foo
-rwSr--r-- 1 sarwar faculty 14 Aug 4 23:38 cp.new
-r-sr-xr-x 1 sarwar faculty 30 Aug 5 05:27 foo
$
```

The `chmod` command with the following syntax may be used to set the SUID bit.

SYNTAX

```
chmod 4xxx file-list
chmod u+s file-list
```

Purpose: Change/set the SUID bit for files in **file-list**

Here, `xxx` is the octal number that specifies the read, write, and execute permissions for user, group, and others, and the octal digit 4 (binary 100) is used to set the SUID bit. When the SUID bit is set, the execute bit for the user is set to `s` (lowercase) if the execute permission is already set for the user; otherwise, it is set to `S` (uppercase). The following session illustrates the use of these command syntaxes. The first, the `ls -l cp.new` command, is used to show that the execute permission for the **cp.new** file is set. The `chmod 4710 cp.new` command is used to set the SUID bit and other nine bits of permission to octal 710. The second, the `ls -l cp.new` command, shows that the `x` bit value has changed to `s`. The two subsequent `chmod` commands are used to set the SUID and execute bits to 0. The `ls -l cp.new` command is used to show that execute permission has been taken away from the owner. The `chmod u+s cp.new` command is used to set the SUID bit again, and the last `ls -l cp.new` command shows that the bit value is `S` (uppercase) because the execute bit was not set prior to setting the SUID bit.

```
$ ls -l cp.new
-rwx--x--- 1 sarwar faculty 14 Aug 4 23:26 cp.new
$ chmod 4710 cp.new
$ ls -l cp.new
-rws--x--- 1 sarwar faculty 14 Aug 4 23:26 cp.new
$ chmod u-s cp.new
$ chmod u-x cp.new
$ ls -l cp.new
-rw---x--- 1 sarwar faculty 14 Aug 4 23:26 cp.new
$ chmod u+s cp.new
$ ls -l cp.new
-rwS--x--- 1 sarwar faculty 14 Aug 4 23:26 cp.new
$
```

Although the idea of the SUID bit is sound, it can compromise the security of the system if not implemented correctly. For example, if the permissions of any set-UID program are set to allow write privileges to others, you can change the program in this file or overwrite the existing program with another program. Doing so would allow you to execute your (new) program with superuser privileges.

5.6.2 Set-Group-ID (SGID) Bit

The SGID bit works in the same manner in which the SUID bit does, but it causes the access permissions of the process to take the group identity of the group to which the owner of the file belongs. This feature is not as dangerous as the SGID feature, because most privileged operations require superuser identity, regardless of the current group ID. The SGID bit is enabled if the execute bit for the group is `s` (or S). If both SGID and execute bits are enabled, the bit is displayed as `s` in the output of the `ls –l` command. If SGID bit is enabled but the execute bit is disabled, the bit is displayed as `S` by the `ls –l` command. The following session shows an example of each case.

```
$ ls -l cp.new foo
-rw-r-Sr-- 1 sarwar faculty 14 Aug 4 23:38 cp.new
-r-xr-sr-x 1 sarwar faculty 30 Aug 5 05:27 foo
$
```

Using either of the following two command syntaxes can set the SGID bit.

SYNTAX

```
chmod 2xxx file-list
chmod g+s file-list
```

Purpose: Change/set the SGID bit for files in **file-list**

Here, `xxx` is the octal number specifying the read, write, and execute permissions for the files in **file-list**, and the octal digit 2 (binary 010) specifies that the SGID bit is to be set for the same files. The following session on PC-BSD illustrates the use of these command syntaxes. The command `chmod 2751 cp.new` sets the SGID bit for the **cp.new** file and sets its access privileges to 751 (`rwxr-x--x`). The rest of the commands are similar to those in Section 5.6.1.

```
$ ls -l cp.new
-rwxr-x--x 1 sarwar faculty 14 Aug 4 23:26 cp.new
$ chmod 2751 cp.new
$ ls -l cp.new
-rwxr-s--x 1 sarwar faculty 14 Aug 4 23:26 cp.new
$ chmod g-s cp.new
$ chmod g-x cp.new
$ ls -l cp.new
-rwxr----x 1 sarwar faculty 14 Aug 4 23:26 cp.new
```

```
$ chmod g+s cp.new
$ ls -l cp.new
-rwxr-S--x 1 sarwar faculty 14 Aug 4 23:26 cp.new
$
```

You can set or reset the SUID and SGID bits by using a single `chmod` command. Thus, the command `chmod ug+s cp.new` can be used to perform this task on the **cp.new** file. You can also set the SUID and SGID bits along with the access permissions bits (read, write, and execute) by preceding the octal number for access privileges by 6 because the leftmost octal digit 6 (110) specifies that both the SUID and SGID bits be set. Thus, the command `chmod 6754 cp.new` may be used to set the SUID and SGID bits for the **cp.new** file and its access privileges to 754.

5.6.3 Sticky Bit

The last of the 12 access bits, the sticky bit, is on if the execute bit for others is `t` (or `T`), as in the case **/tmp** shown below.

```
$ ls -l / | grep tmp
drwxrwxrwt 66 root root 86 Aug 5 05:56 tmp
$
```

The sticky bit can be set for a directory to ensure that an unprivileged user cannot remove or rename files of other users in that directory. You must be the owner of a directory or have appropriate permissions to set the sticky bit for it. Some systems do not allow nonsuperusers to set the sticky bit. It is commonly set for shared directories that contain files owned by several users.

The output of the above command shows that the **/tmp** directory, owned by **root**, has the sticky bit on. It has read, write, and execute permissions for everyone. It means that any user can create a file or directory in **/tmp**. However, because the sticky bit is on, no user (other than the superuser) can remove a file or directory that he/she does not own. Thus, UNIX allows you to remove any files in **/tmp** that you own, but no other file or directory, because you are not the owner of those files and directories.

Originally, this bit was designed to inform the kernel that the code segment of a program is to be shared or kept in the main memory or the *swap space* owing to the frequent use of the program. Thus, when this bit is set for a program, the system tries to keep the executable code for the program (the code segment only) in memory after it finishes execution—the processes literally "stick around" in the memory. If, for some reason, memory space occupied by this program is needed by the system for loading another program, the program with the sticky bit on is saved in the swap space (a special area on the disk used to save processes temporarily). That is, if the sticky bit is set for a program, the code segment of the program is either kept in memory or in the swap space after it finishes its execution. When this program is executed again, with the program in memory, its execution starts right away. If the program is in the swap space, the time needed for loading it is much shorter than if it were stored on disk as a UNIX file. The advantage of this scheme, therefore, is that if a program with the sticky bit on is executed frequently, it is executed much more quickly.

This facility is useful for programs such as compilers, assemblers, editors, and commands such as `ls` and `cat`, which are frequently used in a typical computer system environment. However, care must be taken that not too many programs have this bit set. Otherwise, system performance will suffer because of the lack of free space, with more and more space being used by the programs whose sticky bit is set. This historical use of the sticky bit is no longer needed in newer UNIX systems (starting with 4.4BSD) because virtual memory systems use page replacement algorithms that do not remove recently used program pages/segments. Thus, PC-BSD does not allow you to set the sticky bit on nondirectory files, as shown in the shell session at the end of this section.

Either of the following syntaxes may be used to set the sticky bit.

SYNTAX

```
chmod 1xxx file-list
chmod +t file-list
```

Purpose: Change/set the sticky bit for files in **file-list**

Here, `xxx` is the octal number specifying the read, write, and execute permissions, and the octal digit 1 (binary 001) specifies that the sticky bit is to be set. When the sticky bit is set, the execute bit for others is set to `t` if others already has execute permission; otherwise, it is set to `T`. The following session on PC-BSD illustrates the use of these command syntaxes. The command `chmod 1751 cp.new` tries to set the sticky bit for the **cp.new** file and set its access privileges to 751, but fails. Similarly, the command `chmod +t cp.new` tries to set the sticky bit for the **cp.new** file, but fails. As the outputs of these commands show, PC-BSD does not allow you to enable the sticky bit for a nondirectory file. It does allow you to set this bit for **dir1**, a directory, with the command `chmod +t dir1`. Solaris, on the other hand, would allow you to set the sticky bit for nondirectory files as well as directories. The explanations for the lowercase `t` and uppercase `T` in the following session are the same as for the lowercase `s` and uppercase `S` for SUID and SGID bits.

```
$ ls -l cp.new
-rw-rw-rw- 1 sarwar sarwar 14 Aug 4 23:38 cp.new
$ ls -ld dir1
drwxr-xr-x 2 sarwar sarwar  2 Aug 5 07:59 dir1
$ chmod 1751 cp.new
chmod: cp.new: Inappropriate file type or format
$ chmod +t cp.new
chmod: cp.new: Inappropriate file type or format
$ chmod +t dir1
$ ls -ld dir1
drwxr-xr-t 2 sarwar sarwar  2 Aug 5 07:59 dir1
```

```
$ chmod o-x dir1
$ ls -ld dir1
drwxr-xr-T 2 sarwar sarwar  2 Aug 5 07:59 dir1
$
```

EXERCISE 5.6

Run all the shell sessions shown and discussed in this section to make sure that they work on your system too and to understand how the `chmod` command is used to set and reset special access permission bits (SUID, SGID, and sticky) on your UNIX system. Also, find out if the SGID and sticky bit may be set on your system for both files and directories.

SUMMARY

A time-sharing system has to ensure protection of one user's files from unauthorized (accidental or malicious) access by other users of the system. UNIX provides several mechanisms for this purpose, including one based on access permissions. Files can be protected by informing the system of the type of operations (read, write, and execute) that are permitted on the file by the owner, group (the users who are in the same group as the owner), and others (everyone else on the system). UNIX allows a user to be part of multiple groups. Only the system administrator, also known as the superuser in UNIX jargon, can add you to a group or remove you from a group. You can display the groups you (or any user) are a member of by using the `groups` command. These nine commonly used access permissions are represented by bits. This information is stored in the inode of the file. When a user tries to access a file, the system allows or disallows access based on the file's access privileges stored in the inode.

Access permissions for files can be viewed by using the `ls -l` command. When used with directories, this command displays attributes for all the files in the directories. The `ls -ld` command can be used to view access permissions for directories. The owner of a file can change access privileges on it by using the `chmod` command. The `umask` command, which is usually placed in the **~/.profile** file for System V UNIX and the **~/.login** or **~/.profile** file for BSD UNIX, allows the user to specify a bit mask that informs the system of access permissions that are disabled for the user, group, and others. When a file is created by the UNIX system, it sets access permissions for the file according to the bit mask and the default access permissions for directories, executable files, and text files. File access permissions assigned to newly created files (`A`) are given by the expression `A=B AND C'`, where `B` is the default privileges for the file, `C` is the value of the bit mask, and `AND` is the Boolean function for bitwise `AND`. In a typical system, the mask is set to 022 for default permissions on newly created files and directories. For a fully secure system, it should be set to 077. You can use the `umask` command to display the value of current flag in octal and `umask –S` command to display the flag in a symbolic form.

UNIX also allows three additional bits—the set-user-ID (SUID), set-group-ID (SGID), and sticky bit—to be set. The SUID and SGID bits allow the user to execute commands, including `passwd`, `ls`, `mkdir`, and `ps`, which access important system resources to which

Bit: 12	11	10	9	8	7	6	5	4	3	2	1
SUID	SGID	Sticky	r	w	x	r	w	x	r	w	x
Bits for special access privileges			Owner's access privileges			Group's access privileges			Others' access privileges		

FIGURE 5.2 Position of access privilege bits for UNIX files as specified in the `chmod` command.

access is not allowed otherwise. The sticky bit can be set for a directory to ensure that an unprivileged user cannot remove or rename files of other users in that directory. Only the owner of a directory, or someone else having appropriate permissions, can set the sticky bit for the directory. It is commonly set for shared directories that contain files owned by several users, such as **/tmp** is the UNIX file system structure. Historically, the sticky bit has served another purpose. It can be set for frequently used utilities so that UNIX keeps them in the main memory or on a fixed area on the disk, called the swap space, after their use. This feature makes subsequent access to these files much faster than if they were to be loaded from the disk as normal files. However, due to the implementation of virtual memory systems using advanced algorithms of demand paging, demand segmentation, and paged segmentation on most modern multiuser time-sharing systems, the use of the sticky bit to keep a program memory or swap space resident is no longer required. Its use for directories remains useful and necessary. Therefore, all modern UNIX systems support the sticky bit for directories, but support for the sticky bit for files is no longer maintained across the board. For example, PC-BSD does not provide support for the sticky bit for non-directory files, but Solaris does.

The final format of the 12 access permissions bits, as used in the `chmod` command, is shown in Figure 5.2.

QUESTIONS AND PROBLEMS

1. What are the three basic file protection schemes available in UNIX?
2. List all possible two-letter passwords comprising digits and punctuation letters.
3. If a computer system allows six-character passwords comprising a random combination of decimal digits and punctuation marks, what is the maximum number of passwords that a user will have to try with the brute force method of breaking into a user's account? Why?
4. What is the maximum number of passwords that can be formed if a system allows digits, uppercase and lowercase letters, and punctuation marks to be used? Assume that passwords must be 12 characters long.
5. Suppose that a hacker is trying to guess a password—consisting of eight characters—using uppercase letters, lowercase letters, and digits. Further, suppose that the system forces a 5 s delay after each password guess. How long will it take the hacker to guess

the password in the worst-case analysis? Repeat the exercise if we are allowed to use five punctuation marks as characters in a password. Why? Show all your work.

6. How does file protection based on access permissions work? Base your answer on various types of users of a file and the types of operations they can perform. How many permission bits are needed to implement this scheme? Why?
7. How do the read, write, and execute permissions work in UNIX? Illustrate your answer with some examples.
8. How many user groups exist on your system? How did you get your answer? What groups are you a member of and what is your default group? How many groups is root a member of on your system? How did you obtain your answer? If you used any commands, show the commands and their outputs.
9. Create a file **test1** in your present working directory and set its access privileges to read and write for yourself, read for the users in your group, and none to everyone else. What command did you use to set privileges? Give another command that would accomplish the same thing.
10. The user **sarwar** sets access permissions to his home directory by using the command `chmod 700 $HOME`. If the file **cp.new** in his home directory has read permissions of 777, can anyone read this file? Why or why not? Explain your answer.
11. What is the effect of each command? Explain your answers.
    a. `chmod 776 ~/lab5`
    b. `chmod 751 ~/lab?`
    c. `chmod 511 *.c`
    d. `chmod 711 ~/*`
    e. `ls –l`
    f. `ls –ld`
    g. `ls –l ~/personal`
    h. `ls –ld ~/personal`
12. What does the execute permission mean for a directory, a file type for which the execute operation makes no sense? Explain with an example.
13. Create a file **dir1** in your home directory and use `cp /etc/passwd dir1/mypasswd` command to copy the **/etc/passwd** file into it. Use the `chmod` command to have only the search permission on for it and execute the following commands. What is the result of executing these commands? Do the results make sense to you? Explain.

a. `cd dir1`

b. `ls`

c. `rm dir1/mypasswd`

d. `cp /etc/passwd dir1`

14. What `umask` command should be executed to set the permissions bit mask to 037? With this mask, what default access privileges are associated with any new file that you create on the system? Why? Where would you put this command so that every time you log on to the system this mask is effective?

15. Give a command line for setting the default access mode so that you have read, write, and execute privileges, your group has read and execute permissions, and all others have no permission for a newly created executable file or directory. How would you test it to be sure that it works correctly?

16. Give `chmod` command lines that perform the same tasks as the `mesg n` and `mesg y` commands. (*Hint*: Every hardware device, including your terminal, has an associated file in the **/dev** directory.)

17. What are the purposes of the set-user-ID (SUID), set-group-ID (SGID), and sticky bits?

18. Give one command line for setting all three special access bits (SUID, SGID, and sticky) for the file **cp.new**. (*Hint*: Use octal mode.)

19. In a UNIX system, the `cat` command is owned by root and has its SUID bit set. Do you see any problems with this setup? Explain your answer.

20. Some UNIX systems do not allow users to change their passwords with the `passwd` command. How is this restriction enforced? Is it a good or bad practice? Why?

21. Calculate the file access permissions assigned to newly created directories, executable files, and text files for bit mask 027. Show all your work.

22. Describe briefly the purpose of each of the following commands. Run these commands on your system after creating **~/prog1** (a file) and **~/dir1** (a directory) on it and show the outputs of the commands to verify your answers. If your system does not support any command on your system, explain why you think it does so.

a. `chmod 4776 ~/prog1`

b. `chmod 1776 ~/prog1`

c. `chmod 6776 ~/prog1`

d. `chmod g+s ~/prog1`

e. `chmod +t ~/prog1`

f. `chmod +t ~/dir1`

g. `chmod ugo-r ~/prog1`

h. `chmod a-rw ~/prog1`

i. `chmod ug+x ~/dir1`

j. `chmod go= ~/dir1`

k. `chmod u=rwx ~/prog1`

l. `chmod g=o ~/dir1`

m. `chmod o-wx ~/dir1`

CHAPTER 6

Basic File Processing

Objectives

- To discuss how to display contents of a file
- To explain copying, appending, moving/renaming, and removing/deleting files
- To describe how to determine the size of a file
- To discuss commands for comparing files
- To describe how to combine files
- To discuss printer control commands
- To cover the commands and primitives

  ```
  >, >>, ^, , [ ], *, ?, cancel, cat, cp, diff, head, lp, lpc,
  lpq, lpr, lprm, lpstat, lptest, less, ls, more, mv, nl, pg,
  pr, rm, tail, uniq, wc
  ```

6.1 INTRODUCTION

This chapter describes how some basic file operations can be performed in UNIX. These operations are primarily for nondirectory files, although some are applicable to directories as well; we previously discussed the most commonly used directory operations in Chapter 4. When discussing the file operations in this chapter, we also describe related commands and give examples to illustrate how these commands can be used to perform the needed operations. Remember, complete information on a particular command is available via the `man` command.

6.2 VIEWING CONTENTS OF TEXT FILES

You view files to identify their contents. You can use several UNIX commands to display contents of text files on the display screen. These commands differ from each other in terms of the amount of the file content displayed, the portion of file contents displayed (initial, middle, or last part of the file), and whether the file's contents are displayed one screen or one page at a time. Recall that you can view only those files for which you have the read permission. In addition, you must have the execute (search) permissions for all the directories involved in the pathname of the file to be displayed. Viewing does not mean edit, write, or update—just view.

6.2.1 Viewing Complete Files

You can display the complete contents of one or more files on screen by using the `cat` command. However, because the command does not display file contents one screen or one page at a time, you see only the last page of a file that is larger than one page (i.e., one screen) in size. The following is a brief description of the `cat` command.

SYNTAX

`cat [options] [file-list]`

Purpose: Concatenate/display the files in **file-list** on standard output (screen by default), one after another

Output: Contents of the files in **file-list** displayed on the screen, one file at a time

Commonly used options/features:

- `-e` Display **$** at the end of each line; works in conjunction with the **`-v`** option
- `-n` Put line numbers with the displayed lines
- `-t` Display tabs as **`^I`** and form feeds as **`^L`**
- `-v` Display nonprintable characters, except for the tab, form feed, and newline characters

Here, **file-list** is an optional argument that consists of the pathnames for one or more files, separated by spaces. For example, the following command displays the contents of the **student_records** file in the present working directory. If the file is larger than one page, the file contents quickly scroll off the display screen.

```
% cat student_records
Jonh  Doe   ECE   3.54
Pam   Meyer CS    3.61
Jim   DAVIS CS    2.71
Jason Kim   ECE   3.97
Amy   Nash  ECE   2.38
%
```

The following command displays the contents of files **lab1** and **lab2** in the directory **~/courses/ee446/labs**. The command does not pause after displaying the contents of **lab1**.

```
$ cat ~/courses/ee446/labs/lab1 ~/courses/ee446/labs/lab2
[ contents of lab1 and lab2 ]
$
```

As discussed in Chapter 4, shell metacharacters can be used to specify file names. The contents of all the files in the current directory can be displayed by using the `cat *` command. The `cat exam?` command displays contents of all the files in the current directory starting with the string **exam** and followed by one character such as **exam1**. The contents of all the files in the current directory starting with the string **lab** can be displayed by using the `cat lab*` command.

As indicated by the command syntax, **file-list** is an optional argument. Thus, when the `cat` command is used without any arguments, it takes input from standard input, one line at a time, and sends it to standard output. Recall that, by default, standard input for a command is the keyboard, and standard output is the display screen. Therefore, when the `cat` command is executed without an argument, it takes input from the keyboard and displays it on the screen one line at a time. The command terminates when the user presses `<Ctrl+D>`, the UNIX end-of-file (EOF), on a new line. As is the case throughout the book, the text typed by the user is shown in **bold**.

```
% cat
This is a test.
This is a test.
In this example, the cat command takes input from stdin (keyboard)
In this example, the cat command takes input from stdin (keyboard)
and sends it to stdout (screen), one line at a time.
and sends it to stdout (screen), one line at a time.
However, this is not a typical use of this commend. It is normally
However, this is not a typical use of this commend. It is normally
used to display contents of a file, one line at a time, until it
used to display contents of a file, one line at a time, until it
encounters the end-of-file marker. When the cat command reads
input from stdin,
encounters the end-of-file marker. When the cat command reads
input from stdin,
<Ctrl+D> is the end-of-file marker, as shown below.
<Ctrl+D> is the end-of-file marker, as shown below.
<Ctrl+D>
%
```

At times, you need to view a text file with line numbers. You typically need to do so when, during the software development phase, a compilation of your source code results in compiler errors having line numbers associated with them. The UNIX utility `nl` allows you to display lines of text files with line numbers. Thus, the `nl student _ records` command displays the lines in the **student_records** file with line numbers, as shown in the following session. The `cat -n student _ records` command can also perform the same task.

```
% nl student_records
1     Jonh  Doe   ECE   3.54
```

```
2      Pam    Meyer CS     3.61
3      Jim    DAVIS CS     2.71
4      Jason Kim    ECE    3.97
5      Amy    Nash   ECE    2.38
%
```

Also, if you need to display files with a time stamp and page numbers, you can use the `pr` utility. It displays file contents as the `cat` command does, but it also partitions the file into pages and inserts a header for each page. The page header contains today's date, current time, file name, and page number. The `pr` command, like the `cat` command, can display multiple files, one after the other. The following session illustrates a simple use of the `pr` command.

```
% pr student_records

Aug 25 19:49 2014 student_records Page 1

Jonh   Doe    ECE    3.54
Pam    Meyer CS     3.61
Jim    DAVIS CS     2.71
Jason Kim    ECE    3.97
Amy    Nash   ECE    2.38

<Blank lines until the end of page>

%
```

You can print files with line numbers and a page header by connecting the `nl`, `pr`, and `lp` (or `lpr`) commands. This method is discussed in Chapter 9.

6.2.2 Viewing Files One Page at a Time

If the file to be viewed is larger than one page, you can use the `more` command, also known as the UNIX pager, to display the file a screenful at a time. The following is a brief description of the command.

SYNTAX

`more [options] [file-list]`

Purpose: Concatenate/display the files in **file-list** on standard output a screenful at a time
Output: Contents of the files in **file-list** displayed on the screen, one page at a time
Commonly used options/features:

`+/str`	Start two lines before the first line containing **str**
`-nN`	Display `N` line per screen/page
`+N`	Start displaying the contents of the file at line number `N`

When run without **file-list**, the `more` command, like the `cat` command, takes input from the keyboard one line at a time and sends it to the display screen. If a **file-list** is given as an argument, the command displays the contents of the files in **file-list** one screen at a time. To display the next screen, press `<Space>`. To display the next line in the file, press `<Enter>`. At the bottom left of a screen, the command displays the percentage of the file that has been displayed up to that point. In order to display the next line, you press `<Space>`. To return to the shell, you press the `q` or `Q` key.

The following command displays the **sample** file in the present working directory a screenful at a time. Running this command is equivalent to running the `cat sample | more` command. We discuss the | operator, known as the *pipe* operator, in detail in Chapter 9.

```
$ more sample
[contents of sample]
$
```

The following command displays contents of the files **sample**, **letter**, and **memo** in the present working directory a screenful at a time. The files are displayed in the order they occur in the command.

```
$ more sample letter memo
[contents of sample, letter, and memo]
$
```

The following command displays the contents of the file **param.h** in the directory **/usr/include/sys** one page at a time with 10 lines per page after fully displaying the first page.

```
$ more -n10 /usr/include/sys/param.h
[contents of /usr/include/sys/param.h]
$
```

The following command displays, one page at a time, the contents of all the files in the present working directory that have the **.c** extension (i.e., files containing C source codes).

```
$ more ./*.c
[contents of all .c files in the current directory]
$
```

The `less` command can also be used to view a file page by page. It is similar to the `more` command but is more efficient and has many features that are not available in `more`. It has support for many of the vi and vim Command mode commands. For example, it allows forward and backward movement of file contents one or more lines at a time, redisplaying the screen, and forward and backward string search. It also starts displaying a file without reading the whole file, which makes it more efficient than the `more` command or the vi or vim editor for large files.

6.2.3 Viewing the Head or Tail of a File

Having the ability to view the head (some lines from the beginning) or tail (some lines from the end) of a file is useful in identifying the type of data stored in the file. For example, the head operation can be used to identify a *PostScript* file or a *uuencoded* file, which have special headers, and the tail information could be used to inspect status information at the end of a log file or error file. (We discuss encoding and decoding of files in Chapter 7.) The UNIX commands for displaying the beginning lines or ending lines of a file are `head` and `tail`. The following is a brief description of the `head` command.

SYNTAX

`head [option] [file-list]`

Purpose: Display the initial portions (i.e., heads) files in **file-list**; the default head size is 10 lines

Output: Heads of the files in **file-list** displayed on the monitor screen

Commonly used options/features:

`-N` Display first `N` lines

Without any option and the **file-list** argument, the command takes input from standard input (the keyboard by default). The following session illustrates use of the `head` command. The `cat sample` command is used to display the contents of the **sample** file. The `head sample` command displays the first 10 lines of the **sample** file. The `head -5 sample` command displays the first 5 lines of **sample**.

```
$ cat sample
Ann
Ben
Chen
David
Eto
Fahim
George
Hamid
Ira
Jamal
Ken
Lisa
Mike
Nadeem
Oram
Paul
Queen
Rashid
Srini
Tang
Ursula
Vinny
Wang
X Window System
```

```
Yen
Zen
$ head sample
Ann
Ben
Chen
David
Eto
Fahim
George
Hamid
Ira
Jamal
$ head -5 sample
Ann
Ben
Chen
David
Eto
$
```

You can display heads of multiple files by specifying them as arguments of the `head` command. For example, the `head sample memo1 phones` command displays the first 10 lines each of the **sample**, **memo1**, and **phones** files. The head of each file is preceded by `==> filename <==` at the top left.

The following command, which displays the first 10 lines of the file **otto**, shows that the file is a PostScript file. The output of the command gives additional information about the file, including the name of the software used to create it, the total number of pages in the file, and the page orientation. All of this information is important to know before the file is printed.

```
$ head otto
%!PS-Adobe-3.0
%%BoundingBox: 54 72 558 720
%%Creator: Mozilla (Firefox) HTML->PS
%%DocumentData: Clean7Bit
%%Orientation: Portrait
%%Pages: 1
%%PageOrder: Ascend
%%Title: Otto Doggie
%%EndComments
%%BeginProlog
$
```

Similarly, the following command shows that **data** is a uuencoded file and that, when uudecoded (see Chapter 7), the original file will be stored in the file **data.99**.

```
$ head -4 data
begin 600 data.99 M.0I$3T4L($IO92!#.B`@,#`P.3`P.3H@0T4Z("`@("`@
("!34CH@9&]E,4!S M;6EL92YC;VTZ(#4P,RXR,C(N,C(R,CH@-
3`S+C,S,RXS,S,S"E-A<G=A<BP@ M4WEE9"!-.C`P,#$P-34Z144Z4U(Z<V%R=V%R0
'5P+F5D=3HU,#,N,3$Q+C$R
$
```

The `tail` command is used to display the last portion (tail) of one or more files. It is useful to ascertain, for example, that a PostScript file has a proper ending or that a uuencoded file has the required end on the last line. The following is a brief description of the command.

SYNTAX

`tail [option] [file-list]`

Purpose: Display the last portions (i.e., tails) of files in **file-list**; the default tail size is 10 lines

Output: Tails of the files in **file-list** displayed on the monitor screen

Commonly used options/features:

`-f`	Follow growth of the file after displaying its tail, and display lines as they are appended to the file. The tail command run with this option is terminated with `<Ctrl+C>`.
`±n`	Start **n** lines from the beginning of the file for **+n**, and **n** line (or **n** units) before the end of the file for **−n**; by default, **n** is 10
`-n N`	Display first `N` lines
`-r`	Display lines in the reverse order (last line first)

Like the `head` command, the `tail` command takes input from standard input if no **file-list** is given as an argument. The following session illustrates how the `tail` command can be used with and without options. We use the same **sample** file that we used for the `head` command. The `tail sample` command displays the last 10 lines (the default tail size) of the **sample** file, and the `tail -5 sample` displays the last five lines of the **sample** file. The `tail +12 sample` command displays the tail of the file starting with line number 12 (not the last 12 lines). Finally, the `tail -5r sample` command displays the last 5 lines of the **sample** file in reverse order.

```
$ tail sample
Queen
Rashid
Srini
Tang
Ursula
Vinny
Wang
X Window System
Yen
```

```
Zen
$ tail -5 sample
Vinny
Wang
X Window System
Yen
Zen
$ tail +12 sample
Lisa
Mike
Nadeem
Oram
Paul
Queen
Rashid
Srini
Tang
Ursula
Vinny
Wang
X Window System
Yen
Zen
$ tail -5r sample
Zen
Yen
X Window System
Wang
Vinny
$
```

The following commands show that files **otto** and **data** have proper PostScript and uuencoded tails.

```
$ tail -5 otto
8 f3
( ) show
pagelevel restore
showpage
%%EOF
$ tail data
M;W4@:&%V90IN;W0@=')I960@;W5T(&9O<B!L;VYG('1I;64N("!(;W=E=F5R
M+"!T;R!B92!S=6-C97-S9G5L+"!Y;W4@;75S="!T<@I(96QL;RP@5V]R;&0A
"(0H`
`
end
$
```

The -f option of the tail command is very useful if you need to see the tail of a file that is growing. This situation occurs quite often when you run a simulation program that takes a long time to finish (several minutes, hours, or days) and you want to see the data produced by the program as it is generated. It is convenient to do so if your UNIX system runs the X Window System (see Chapter 22). In an X environment, you can run the tail command in an *xterm* (a console window) to monitor the newly generated data as it is generated and keep doing your other work concurrently. The following command displays the last 10 lines of the **sim.data** file and displays new lines as they are appended to the file. You can terminate the command by pressing <Ctrl+C>.

```
$ tail -f sim.data
... last 10 lines of sim.data ...
... more data as it is appended to sim.data ...
```

Sometimes, while identifying problems in a UNIX system, the system administrator needs to display files in the **/var** directory that keep growing because the kernel and applications keep appending new messages to them, including files in the **/var/spool**, **/var/mail**, and **/var/log** directories. The system administrators are able to view these files as they are appended using the tail –f command, as in the following session to display the last 10 lines in the **/var/log/messages** file and to continue to show new messages from the kernel and applications as they are appended to this file.

```
$ tail -f /var/log/messages
... last 10 lines of /var/log/messages ...
... more data as it is appended to /var/log/messages ...
```

In the following in-chapter exercises, you are asked to use the cat, head, more, pr, and tail commands for displaying different parts of text files, with and without page titles and numbers.

EXERCISE 6.1

Insert the **student_records** file used in Section 6.2.1 in your current directory. Add to it 10 more students' records. Display the contents of this file by using the cat student_records and cat -n student_records commands. What is the difference between the outputs of the two commands?

EXERCISE 6.2

Display the **student_records** file by using the more and pr commands. What command lines did you use?

EXERCISE 6.3

Display the **/etc/passwd** file two lines before the line that contains your login name. What command line did you use?

EXERCISE 6.4

Give commands for displaying the first and last seven lines of the **student_records** file.

6.3 COPYING, MOVING, AND REMOVING FILES

In this section, we describe commands for performing copy, as well as move/rename and remove/delete operations on files in a file structure. The commands discussed are `cp`, `mv`, and `rm`.

6.3.1 Copying Files

The UNIX command for copying files is `cp`. The following is a brief description of the command.

SYNTAX

```
cp [options] file1 file2
```

Purpose: Copy **file1** to **file2**. If **file2** is a directory, make a copy of **file1** in this directory.

Commonly used options/features:

- `-f` Force copying if there is no write permission on **file2**
- `-i` If **file2** exists, prompt before overwriting
- `-p` Preserve file attributes such as owner ID, group ID, permissions, and modification times
- `-r` Recursively copy files and subdirectories

You must have permission to read the source file (**file1**) and permission to execute (search) the directories that contain **file1** and **file2**. In addition, you must have the write permission for the directory that contains **file2** if it does not already exist. If **file2** exists, you don't need the write permission to the directory that contains it, but you must have the write permission to **file2**. If the destination file (**file2**) exists, by default, it will be overwritten without informing you if you have permission to write to the file. To be prompted before an existing file is overwritten, you need to use the `-i` option. If you do not have permission to write to the destination file, you will be informed of this. If you do not have permission to read the source file, an error message will appear on your screen.

The following `cp` command line makes a copy of **temp** in **temp.bak**. The `ls` commands show the state of the current directory before and after execution of the `cp` command. Figure 6.1 shows the same information in pictorial form.

```
$ ls
memo   sample      temp
$ cp temp temp.bak
$ ls
memo   sample      temp   temp.bak
$
```

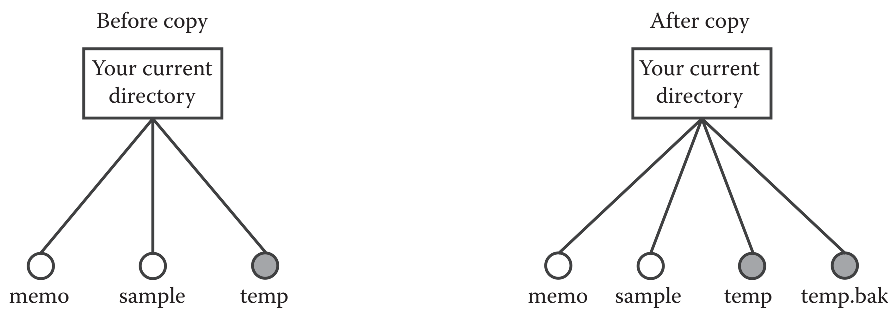


FIGURE 6.1 State of the current directory before and after the temp file has been copied to **temp.bak**.

The command returns an error message if **temp** does not exist or if it exists but you do not have permission to read its content. The command also returns an error message if **temp.bak** exists and you do not have permission to write to it. The following session illustrates these points. The first error message is reported because the **letter** file does not exist in the current directory. The second error message is reported because you do not have permission to read the **sample** file. The last command reports an error message because **temp.bak** exists and you do not have write permission for it. You can override the absence of write permission and force copying by using the `–f` option, as shown in the next command. The `ls –l memo temp.bak` command is used to show that the copying has actually taken place; that is, the data has been copied, but the time stamp for the file is the current time. If you want to copy both the data and attributes of the source file, you need to use the `cp` command with `–f` and `–p` options, as in the last `cp` command that follows. The last `ls –l memo temp.bak` command is used to show that both data and file attributes such as the time stamp have been copied.

```
% ls -l
total 3
-rwxr----- 1 sarwar faculty 371 Aug 28 07:01 memo
--wxr----- 1 sarwar faculty 164 Jul 25 12:35 sample
-r-xr----- 1 sarwar faculty 792 Aug 28 07:01 temp
-r-xr----- 1 sarwar faculty 792 Aug 28 07:05 temp.bak
% cp letter letter.bak
cp: letter: No such file or directory
% cp sample sample.new
cp: sample: Permission denied
% cp memo temp.bak
cp: temp.bak: Permission denied
% cp -f memo temp.bak
% ls -l memo temp.bak
-rwxr----- 1 sarwar faculty 0 Aug 28 07:01 memo
-rwxr----- 1 sarwar faculty 0 Aug 28 07:22 temp.bak
% cp –fp memo temp.bak
% ls -l memo temp.bak
```

```
-rwxr----- 1 sarwar faculty 0 Aug 28 07:01 memo
-rwxr----- 1 sarwar faculty 0 Aug 28 07:01 temp.bak
%
```

The following command makes a copy of the **.profile** file in your home directory and puts it in the **.profile.old** file in the **sys.backups** subdirectory, also in your home directory. This command works regardless of the directory you are in when you run the command because the pathname starts with your home directory. You should execute this command before changing your runtime environment specified in the ~/**.profile** file, so that you have a backup copy of the previous working environment in case something goes wrong when you set up the new environment. The command produces an error message if ~/**.profile** does not exist, if you do not have permission to read it, if the ~/**sys.backups** directory does not exist or you do not have the execute (search) and write permissions for it, or if **.profile.bak** exists but you do not have permission to read it.

```
% cp ~/.profile ~/sys.backups/.profile.bak
%
```

The following command copies all the files in the current directory, starting with the string lab to the directory ~/**courses/ee446/backups**. The command also prompts you for overwriting if any of the source files already exist in the backups directory. In this case (in which multiple files are being copied), if **backups** is not a directory, or if it does not exist, an error message is displayed on the screen informing you that the target must be a directory.

```
% cp -i lab* ~/courses/ee446/backups
%
```

If you want to copy a complete directory to another directory, you need to use the `cp` command with the `-r` option. This option recursively copies files and subdirectories from the source directory to the destination directory. It is a useful option that you can use to create backups of important directories periodically. Thus, the following command recursively copies the ~/**courses** directory to the ~/**backups** directory.

```
$ cp -r ~/courses ~/backups
$
```

This command creates copies the ~/**courses** directory, including all the files and directory hierarchies under the ~/**courses** directory, and places it under the ~/**backups** directory. Figure 6.2 shows the state of your home directory (~) before and after the execution of the command.

6.3.2 Moving Files

Files can be moved from one directory in a file structure to another. This operation in UNIX can also result in simply renaming a file if it is on the same file system. The renaming

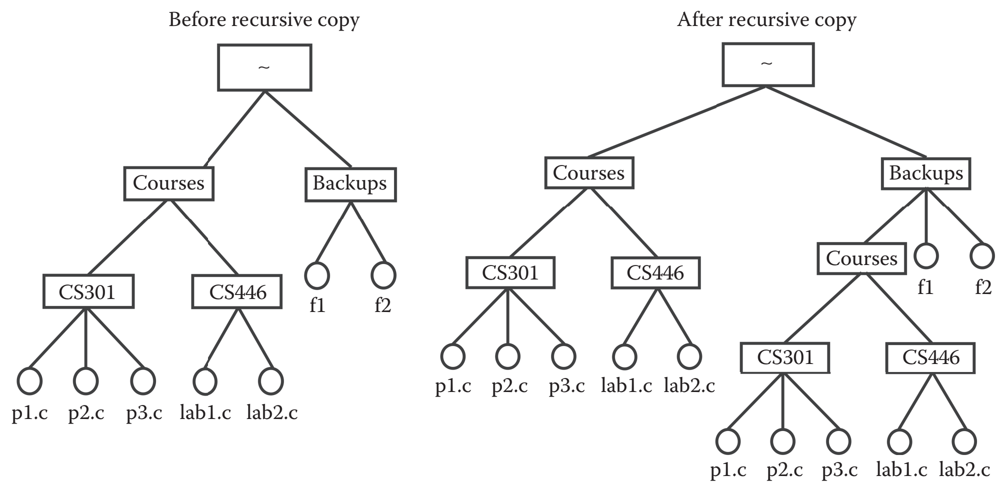


FIGURE 6.2 Current directory before and after the `cp -r ~/courses ~/backups` command.

operation is equivalent to creating a hard link (see Chapter 8) to the file, followed by removing/deleting (see Section 6.3.3) the original file. If the source and destination files are on different file systems, the move operation results in a physical copy of the source file to the destination, followed by removal of the source file. The command for moving files is `mv`. The following is a brief description of the command.

SYNTAX

```
mv [options] file1 file2
mv [options] file-list directory
```

Purpose: First syntax: Move **file1** to **file2** or rename **file1** to **file2**
Second syntax: Move all the files in **file-list** to **directory**

Commonly used options/features:

`-f` Force move regardless of the permissions of the destination file
`-i` Prompt the user before overwriting the destination file

You must have the write and execute access permissions for the directory that contains the source file (**file1** in the description), but you do not need to have the read, write, or execute permissions for the file itself. Similarly, you must have the write and execute permissions for the directory that contains the target/destination file (**file2** in the description), execute permission for every directory in the pathname for the destination file, and write permission for the destination file if it already exists. If the destination file exists, by default, it is overwritten without informing you. If you use the `-i` option, you are prompted before the destination file is overwritten.

The following command moves **temp** to **temp.moved**. In this case, the **temp** file is renamed **temp.moved**. The `mv` command returns an error message if **temp** does not exist, or if you do not have the execute permission for the directory it is in. The command prompts you for moving the file if **temp.moved** already exists, but you do not have write permission for it.

```
$ mv temp temp.moved
$
```

The following command moves **temp** to the backups directory as the **temp.old** file. Figure 6.3 shows the state of your current directory before and after the **temp** file is moved.

```
$ mv temp backups/temp.old
$
```

The following command is a sure move; you can use it to force the move, regardless of the permissions for the target file, **temp.moved**.

```
$ mv -f temp temp.moved
$
```

The following command moves all the files and directories (excluding hidden files) in **dir1** to the **dir2** directory. The command fails, and an error message appears on your screen, if **dir2** is not a directory, if it does not exist, or if you do not have the write and execute permissions for it.

```
$ mv dir1/* dir2
$
```

After the command is executed, **dir1** contains the hidden files only. You can use the `ls -a` command to confirm the status of **dir1**.

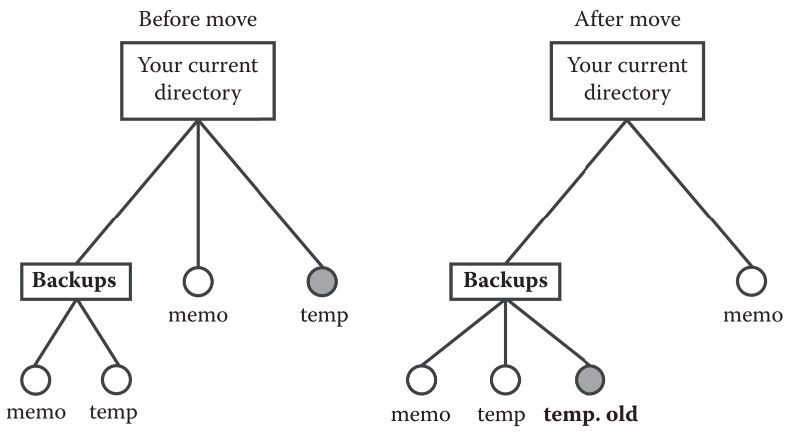


FIGURE 6.3 Current directory before and after the `mv temp backups/temp.old` command.

6.3.3 Removing/Deleting Files

When files are not needed anymore, they should be removed from a file structure to free up disk space to be reused for new files and directories. The UNIX command for removing (deleting) files is `rm`. The following is a brief description of the command.

SYNTAX

`rm [options] file-list`

Purpose: Removes the files in **file-list** from the file structure (and disk)

Commonly used options/features:

- `-f` Force remove regardless of the permissions for **file-list**
- `-i` Prompt the user before removing the files in **file-list**
- `-r` Recursively remove the files in the directory, which is passed as an argument. This removes everything under the directory, so be sure you want to do so before using this option.

If files in **file-list** are pathnames, you need the read and execute permissions for all the directory components in the pathnames and the read, write, and execute permissions for the last directory (that contains the file or files to be deleted). You must also have write permission for the files themselves for their removal without prompting you. If you run the command from a terminal and do not have write permission for the file to be removed, the command displays your access permissions for the file and prompts you for your permission to remove it.

The following command lines illustrate use of the `rm` command to remove one or more files from various directories.

```
$ rm temp
$ rm temp backups/temp.old
$ rm -f phones grades ~/letters/letter.john
$ rm ~/dir1/*
$
```

The first command removes **temp** from your current directory. The second command removes the **temp** file from your current directory and the **temp.old** file from the backups directory in your current directory. Figure 6.4 shows the semantics of this command. The third command removes the **phones**, **grades**, and **~/letters/letter.john** files, regardless of their access permissions. The fourth command removes all the files from the **~/dir1** directory; the directories are not removed.

Now, consider the following commands that use some shell metacharacter features (see Chapter 7).

```
$ rm [kK]*.prn
$ rm [a-kA-Z]*.prn
$
```

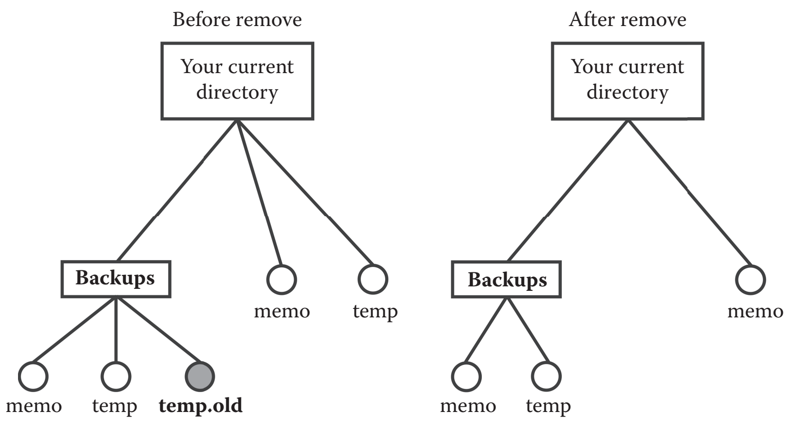


FIGURE 6.4 Current directory before and after execution of `rm temp backups/temp.old` command.

The first command removes all the files in current directory that have the **.prn** extension and names starting with `k` or `K`. The second command removes all the files in the current directory that have the **.prn** extension and names starting with a lowercase letter from `a` through `k` or an uppercase letter.

In Chapter 4, we talked about removing directories and showed that the `rmdir` command can be used to remove only the empty directories. The `rm` command with the `-r` option can be used to remove nonempty directories recursively. Thus, the following command recursively removes the **OldDirectory** in your home directory. This command prompts you if you do not have the permission to remove a file. If you do not want the system to prompt you and you want to force remove the **~/OldDirectory** recursively, then use the `rm –rf ~/OldDirectory` command. This command is one of the commands that you must never execute unless you really know its potentially catastrophic consequences: the loss of all the files and directories in a complete directory hierarchy. But the command is quite useful if you want to free up some disk space.

```
$ rm -r ~/OldDirectory
$
```

You should generally combine the `-i` and `-r` options to remove a directory (**~/OldDirectory** in this case) recursively, as shown in the following command. The `-i` option is for interactive removal, and when you use this option, the `rm` command prompts you before removing a file. This way you can ensure that you do not remove an important file by mistake.

```
$ rm -ir ~/OldDirectory
rm: examine files in directory /home/sarwar/OldDirectory (y/n)? y
rm: remove /home/sarwar/OldDirectory/John.11.14.2013 (y/n)? y
rm: remove /home/sarwar/OldDirectory/Tom.2.24.2011 (y/n)?
...
$
```

6.3.4 Determining File Size

You can determine the size of a file by using one of several UNIX commands. The two commands commonly used for this purpose that are available in all UNIX versions are `ls -l` and `wc`. We described the `ls -l` command in Chapter 5, where we use it to determine the access permissions for files. We revisit this command here in the context of determining file size.

As we mentioned before, the `ls -l` command displays a long list of the files and directories in the directory (or directories) specified as its argument. You must have the read and execute permissions for a directory to be able to run the `ls` command on it successfully; no permissions are needed on the files in the directory to be able to see the list. The command gives output for the current directory if none is specified as an argument. The output of this command has nine fields, and the fifth field gives file sizes in bytes (see Section 5.5). In the following command, the size of the **lab2** file is 709 bytes.

```
$ ls -l lab2
-rw-r--r-- 1 sarwar faculty 709 Apr 5 11:23 lab2
$
```

This command also displays the size of directory files. You can also use it to get the sizes of multiple files by specifying them in the command line and separating them by spaces. For example, the following command shows that sizes of the **lab1** and **lab2** files are 163 bytes and 709 bytes, respectively.

```
$ ls -l lab1 lab2
-rw-r--r-- 1 sarwar faculty 163 Jul 9 16:47 lab1
-rw-r--r-- 1 sarwar faculty 709 Apr 5 11:23 lab2
$
```

The following command uses the shell metacharacter `*` to display the long listings for all the files in the **~/courses/ee446** directory.

```
$ ls -l ~/courses/ee446/*
... output of the command ...
$
```

Whereas `ls -l` is a general-purpose command that can be used to determine many of the attributes of one or more files, including their sizes in bytes/characters, `wc` is a special purpose command that displays only file sizes. The following is a brief description of the `wc` command.

SYNTAX

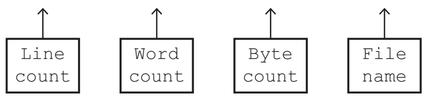
`wc [options] file-list`

Purpose: Display sizes of the files in **file-list** as number of lines, words, and characters

Commonly used options/features:

`-c`	Display only the number of characters
`-l`	Display only the number of lines
`-w`	Display only the number of words

```
% wc sample
      4      44     227   sample
%
```



```
% wc letter sample test
        44  250   1687  letter
         4   44    227  sample
         2   12     90  test
        50  306   2004  total
% wc -c letter sample test
      1687  letter
       227  sample
        90  test
      2004  total
% wc -lw letter sample test
        44  250   letter
         4   44   sample
         2   12   test
        50  306   total
%
```

The first command displays the number of lines, words, and characters in the **sample** file in the present working directory. The size of **sample** is 4 lines, 44 words, and 227 bytes. The second command displays the same information for the files **letter**, **sample**, and **test** in the present working directory. The last line in the output of this command also displays the total line count, word count, and byte count for all three files. The third command displays the number of characters in **letter**, **sample**, and **test**. The last command shows that multiple options can be used in a single command; in this case, the output is the number of words and letters for the three files in the command line.

On FreeBSD, the wc command with a directory argument returns three numbers along with the name of the directory: **0**, **1**, and the number of directory entries in the directory including the **.** (dot) and **..** (dotdot) directories, as in:

```
$ wc /etc
      0     1     116 /etc
$
```

This only applies to directories that contain regular files and directories as their contents. However, this style of output is not produced for directories that contain device (character special and block special) files and other types of files.

The wc command can be used with shell metacharacters such as * and ?. The following command displays sizes of all the files in the directory **/usr/include/sys**.

```
% wc /usr/include/sys/*
      61    332   2213 /usr/include/sys/_bitset.h
      63    320   2150 /usr/include/sys/_bus_dma.h
      64    397   2626 /usr/include/sys/_callout.h
      56    271   1929 /usr/include/sys/_cpuset.h
...
      52    275   1868 /usr/include/sys/_timeval.h
     115    721   4722 /usr/include/sys/_types.h
      74    346   2396 /usr/include/sys/_umtx.h
      51    303   1918 /usr/include/sys/_unrhdr.h
     208    867   7563 /usr/include/sys/aac_ioctl.h
     125    677   4268 /usr/include/sys/acct.h
     416   1720  14861 /usr/include/sys/acl.h
...
%
```

6.4 APPENDING TO FILES

Appending to a file means putting new data at the end of the contents of the file. If the file does not exist, it is created to contain the new data. The append operation is useful when an application or a user needs to augment a file by adding data to it. The following command syntax is used to append one or more files, or keyboard input, at the end of a file.

SYNTAX

```
cat [file-list] >> destination-file
```

Purpose: Append the contents of the files in **file-list**, in the order specified in the command line, at the end of **destination-file**

The >> operator is the UNIX append operator. We discuss the >>, <, and > operators in detail in Chapter 9. That chapter describes how the input of your commands can be read as input from a file instead of the keyboard, and how the output and error messages of your commands can be redirected from the terminal (or console widow) to files. In this chapter, we use these operators only to describe how you can append new data at the end of the current contents of a file and how you can combine the contents of multiple files and put them in one file using the `cat` command.

The following session illustrates how the append operation works. The `cat sample >> temp` command appends the contents of the **sample** file at the end of the **temp** file. The `cat` commands before and after this command show the contents of the files involved. The command syntax can be used to append multiple files to a file, as shown in the command `cat memo1 memo2 memo3 >> memos.record`. This command appends the contents of the **memo1**, **memo2**, and **memo3** files at the end of the **memos.record** file.

```
$ cat temp
This is a simple file used to illustrate the working of append
operation. The new data will be appended right below this line.
$ cat sample
These are the new data that will be appended at the end of the
test file.
$ cat sample >> temp
$ cat temp
This is a simple file used to illustrate the working of append
operation. The new data will be appended right below this line.
These are the new data that will be appended at the end of the
test file.
$ cat memo1 memo2 memo3 >> memos.record
$
```

Without the optional **file-list** argument (see the command description), the command can be used to append keyboard input at the end of **destination-file**. The `cat >> test.letter` command below takes input from the keyboard and appends it to a file called **test.letter**. The command terminates when you press `<Ctrl+D>` on a new line.

```
$ cat test.letter
John Doe
12345  First Lane
Second City, State 98765
$ cat >> test.letter
September 1, 2014
Dear John:
This is to inform you ...
...
<Ctrl+D>
$ cat test.letter
John Doe
12345  First Lane
Second City, State 98765
September 1, 2014
Dear John:
This is to inform you ...
...
$
```

6.5 COMBINING FILES

The following command syntax can be used to combine multiple files into one file.

SYNTAX

```
cat [file-list] > destination-file
```

Purpose: Put the contents of the files in **file-list**, in the order specified in the command line, and put them in **destination-file**

The **destination-file** is overwritten if it already exists. If you do not have the write permission for the **destination-file**, the command displays an error message informing you that you do not have permission to write to the file. Without the optional **file-list** argument, you can use the command to put keyboard input in **destination-file**. Thus, this command syntax can be used to create a new file whose contents are what you enter from the keyboard until you press `<Ctrl+D>` on a new line, as is the case with the `cat >> test.letter` command in the previous session.

The following session illustrates how this command works with arguments. The `ls -l` command is used to view permissions for the files. The `wc memo?` command displays the sizes of all the files in the current directory that start with the string memo and have one or more letters after this string. The third command combines the contents of the **memo1**, **memo2**, and **memo3** files and puts them in the **memos.y2k14** file in the order they appear in the command. The `wc memos.y2k14` command is used to confirm that the **memos.y2k14** file has the same number of lines, words, and characters as the three memo files combined. Execution of the `cat memo1 memo2 memo3 > memos.2014` command shows that you do not have permission to write to **memos.2014**.

```
% ls -l
-r-xr--r-- 1 sarwar faculty 1687 Jan 10 19:15 memo1
-r-xr--r-- 1 sarwar faculty 1227 Feb 19 14:37 memo2
-r-xr--r-- 1 sarwar faculty  790 Sep  1 19:16 memo3
-r-------- 1 sarwar faculty 9765 Jan 15 22:11 memos.2014
% wc memo?
      44    250   3352  memo1
      34    244   4083  memo2
      12    112    907  memo3
      90    606   3704  total
% cat memo1 memo2 memo3 > memos.y2k14
% wc memos.y2k14
      90    606   3704  memos.y2k14
% cat memo1 memo2 memo3 > memos.2014
memos.2014: Permission denied.
%
```

You can also do the task of the `cat memo1 memo2 memo3 > memos.y2k14` by using the following command sequence.

```
$ cat memo1 > memos.y2k14
$ cat memo2 >> memos.y2k14
$ cat memo3 >> memos.y2k14
$
```

The following in-chapter exercises ask you to practice using the `cp`, `mv`, `ls -l`, `wc`, and `cat` commands and the operator for appending to a file.

EXERCISE 6.5

Copy the **.profile** (or **.login** in a BSD UNIX-based system) file in your home directory to a file **.profile.old** (or **.login.old**) in a directory called **backups**, also in your home directory. Assume that you are in your home directory. What command did you use?

EXERCISE 6.6

Create a directory called **new.backups** in your home directory and move all the files in the backups directory to **new.backups**. What commands did you use?

EXERCISE 6.7

Display the size in bytes of a file **lab3** in the **~/ece345** directory. What command did you use?

EXERCISE 6.8

Give a command for appending all the files in the **~/courses/ece446** directory to a file called **BigBackup.ece446** in the **~/courses** directory.

6.6 COMPARING FILES

At times, you will need to compare two versions of a program code or some other document to find out where they differ from each other. You can use the `diff` command to perform this task. The command compares two files and displays differences between them in terms of commands that can be used to convert one file to the other. The following is a brief description of the command.

SYNTAX

```
diff [options] [file1] [file2]
```

Purpose: Compare **file1** with **file2** line by line and display differences between them as a series of commands/instructions for the `ed` editor that can be used to convert **file1** to **file2** or vice versa; read from standard input if – is used for **file1** or **file2**

Commonly used options/features:

- `-b` Ignore trailing (at the end of lines) white spaces (blanks and tabs), and consider other strings of white spaces equal
- `-e` Generate and display a script for the `ed` editor that can be executed to change **file1** to **file2**
- `-h` Do fast comparison (the **`-e`** option may not be used in this case)

TABLE 6.1 File Conversion Instructions Produced by `diff`

Instruction	Description for Changing **file1** to **file2**
`L1aL2,L3` > lines L 2 through L 3	Append lines L2 through L3 from **file2** after line L1 in **file1**

`L1,L2cL3,L4`
< lines L1 through L 2 in file1
`---`
> lines L 3 through L4 in file2
	Change lines L1 through L2 in **file1** to lines L3 through L4 in **file2**

`L1,L2dL3` < lines L1 through L 2 in file1	Delete lines L1 through L2 from **file1**

The **file1** and **file2** arguments can be directories. If **file1** is a directory, `diff` searches it to locate a file named **file2** and compares it with **file2** (the second argument). If **file2** is a directory, `diff` searches it to locate a file named **file1** and compares it with **file1** (the first argument). If both arguments are directories, the command compares all pairs of files in these directories that have the same names.

The `diff` command does not produce any output if the files being compared are the same. When used without any options, the `diff` command produces a series of instructions for the `ed` editor that can be used to convert **file1** to **file2** if the files are different. The instructions are `a` (add), `c` (change), and `d` (delete) and are described in Table 6.1.

The following session illustrates a simple use of the `diff` command.

```
$ cat Fall_OH
Office Hours for Fall 2014
Monday
9:00 - 10:00 A.M.
3:00 - 4:00 P.M.
Tuesday
10:00 - 11:00 A.M.
Wednesday
9:00 - 10:00 A.M.
3:00 - 4:00 P.M.
Thursday
11:00 A.M. - 12:00 P.M.
2:00 - 3:00 P.M.
4:00 - 4:30 P.M.
$ cat Spring_OH
Office Hours for Spring 2015
Monday
9:00 - 10:00 A.M.
3:00 - 4:00 P.M.
Tuesday
10:00 - 11:00 A.M.
1:00 - 2:00 P.M.
Wednesday
9:00 - 10:00 A.M.
Thursday
```

```
11:00 A.M. - 12:00 P.M.
$ diff Fall_OH Spring_OH
1c1
< Office Hours for Fall 2014
---
> Office Hours for Spring 2015
6a7
> 1:00 - 2:00 P.M.
9d9
< 3:00 - 4:00 P.M.
12,13d11
< 2:00 - 3:00 P.M.
< 4:00 - 4:30 P.M.
$
```

The instruction `1c1` asks the `ed` editor to change the first line in the **Fall_OH** file (`Office Hours for Fall 2014`) to the first line in the **Spring_OH** file (`Office Hours for Spring 2015`). The `6a7` instruction asks the `ed` editor to append line 7 in **Spring_OH** after line 6 in **Fall_OH**. The `12,13d11` instruction asks the `ed` editor to delete lines 12 and 13 from **Fall_OH**.

The following session illustrates use of the `-e` option with the `diff` command and how the output of this command can be given to the `ed` editor in order to make **Fall_OH** the same as **Spring_OH**. The command is used to show you what the output of the command looks like. The second `diff` command (with `> diff.script`) is used to save the command output (the `ed` script) in the **diff.script** file. The `cat >> diff.script` command is used to convert the **diff.script** file into a complete working script for the `ed` editor by adding two lines containing w and q. As previously stated, this command terminates with `<Ctrl+D>`. Finally, the `ed` command is run to change the contents of **Fall_OH**, according to the script produced by the `diff -e` command, and make it the same as **Spring_OH**. The numbers 209 and 177 show the sizes of the **Fall_OH** file before and after the execution of the `ed` command. The last command, `diff Fall _ OH Spring _ OH`, is run to confirm that the two files are the same.

```
$ diff -e Fall_OH Spring_OH
12,13d
9d
6a
1:00 - 2:00 P.M.
.
1c
Office Hours for Spring 2015
.
$ diff -e Fall_OH Spring_OH > diff.script
$ cat >> diff.script
w
q
```

```
<Ctrl+D>
$ ed Fall_OH < diff.script
209
177
$ diff Fall_OH Spring_OH
$
```

Most systems have a command called diff3 that can be used to do a three-way comparison; that is, three files can be composed.

6.7 LOCATING AND REMOVING REPETITION WITHIN TEXT FILES

You can use the uniq command to remove all but one copy of the successive repeated lines in a file. The command is intended for files of sorted content although it can work on files without sorted content, as shown in the example. We discuss sorting in Chapter 7. The following is a brief summary of the command.

SYNTAX

uniq [options] [input-file] [output-file]

Purpose: Remove repetitious lines from the sorted **input-file** and send unique (nonrepeated) lines to **output-file**. The **input-file** does not change. If no **output-file** is specified, the output of the command is sent to standard output. If no **input-file** is specified, the command takes input from standard input.

Commonly used options/features:

-c	Precede each output line by the number of times it occurs
-d	Display the repeated lines
-f N	Ignore the first N fields in input lines while doing comparisons
-i	Perform case-sensitive comparison of input lines
-s C	Ignore the first C characters in input lines while doing comparisons
-u	Display the lines that are not repeated

The following session illustrates how the uniq command works. The cat command is used to show the contents of the **sample** file. The uniq sample command shows that only consecutive duplicate lines are considered duplicate. The uniq -c sample command shows the line count for every line in the file. The uniq -d sample command is used to output repeated lines only. Finally, the uniq -d sample out command sends the output of the command to the **out** file. The cat out command is used to show the contents of **out**. Note that the uniq command only works for unsorted files if repeated lines are adjacent.

```
$ cat sample
This is a test file for the uniq command.
It contains some repeated and some nonrepeated lines.
Some of the repeated lines are consecutive, like this.
Some of the repeated lines are consecutive, like this.
Some of the repeated lines are consecutive, like this.
And, some are not consecutive, like the following.
Some of the repeated lines are consecutive, like this.
The above line, therefore, will not be considered a repeated
line by the uniq command, but this will be considered repeated!
line by the uniq command, but this will be considered repeated!
$ uniq sample
This is a test file for the uniq command.
It contains some repeated and some nonrepeated lines.
Some of the repeated lines are consecutive, like this.
And, some are not consecutive, like the following.
Some of the repeated lines are consecutive, like this.
The above line, therefore, will not be considered a repeated
line by the uniq command, but this will be considered repeated!
$ uniq -c sample
      1 This is a test file for the uniq command.
      1 It contains some repeated and some nonrepeated lines.
      3 Some of the repeated lines are consecutive, like this.
      1 And, some are not consecutive, like the following.
      1 Some of the repeated lines are consecutive, like this.
      1 The above line, therefore, will not be considered a
      repeated
      2 line by the uniq command, but this will be considered
      repeated!
$ uniq -d sample
Some of the repeated lines are consecutive, like this.
line by the uniq command, but this will be considered repeated!
$ uniq -d sample out
$ cat out
Some of the repeated lines are consecutive, like this.
line by the uniq command, but this will be considered repeated!
$
```

In the following in-chapter exercises, you will use the `diff` and `uniq` commands to appreciate the tasks they perform.

EXERCISE 6.9

Duplicate the interactive sessions given in Section 6.6 to appreciate how the `diff` command works.

EXERCISE 6.10

Give a command to remove all but one occurrence of the consecutive duplicate lines in a file called **phones** in the **~/personal** directory. Assume that you are not in your home directory.

6.8 PRINTING FILES AND CONTROLLING PRINT JOBS

We briefly discussed the UNIX commands for printing files in Chapter 2. In this section, we cover file printing fully, including commands related to printing and printer control. These commands include commands for printing files, checking the status of print requests/jobs on a printer, and canceling print jobs. We describe commands that are available in both PC-BSD and Solaris.

6.8.1 UNIX Mechanism for Printing Files

The process of printing files is similar to the process of displaying files; in both cases, the contents of one or more files are sent to an output device. In the case of displaying output, the output device is a display screen, whereas in the case of printing output, the output device is a printer. Another key difference results primarily from the fact that every user has an individual display screen but that many users may share a single printer on a typical UNIX (or any time-sharing) system. Thus, when you use the `cat` or `more` command to display a file, the contents of the file are immediately sent to the display screen by UNIX. However, when you print a file, its contents are not immediately sent to the printer because the printer might be busy printing some other file (yours or some other user's). To handle multiple requests, a *first-come first-served* (FCFS) mechanism places a print request in a queue associated with the printer to which you have sent your print request and processes the request in its turn when the printer is available.

UNIX maintains a queue of print requests, called the *print queue*, associated with every printer in the system. Each request is called a *job*. A job is assigned a number, called the *job ID*. When you use a command to print a file, the system makes a temporary copy of your file, assigns a job ID to your request, and puts the job in the print queue associated with the printer specified in the command line. When the printer finishes its current job, it is given the next job from the front of the print queue. Thus, your job is processed when the printer is available and your job is at the head of the print queue.

A UNIX process called the *printer spooler* or *printer daemon* performs the work of maintaining the print queue and directing print jobs to the right printer. This process is called lpd. It starts execution in the background when the system boots up and waits for your print requests. We discuss daemons in Chapter 10, but for now, you can think of a daemon as a process that runs but you are not aware of its presence while it interacts with your terminal.

System V and BSD have different command sets for printing and controlling print jobs. Because many of the contemporary UNIX systems are compatible with both System V and BSD, they contain both sets of commands. Table 6.2 contains a list of the printing-related commands for both systems; all are available in PC-BSD and Solaris. The superuser—the system administrator—normally uses the last two commands, `lpc` and `lptest`.

6.8.2 Printing Files

As shown in Table 6.2, you can print files by using the `lp` command on a System V–compatible UNIX system and the `lpr` command on a BSD-compliant UNIX system. It

TABLE 6.2 List of Commands Related to Printing Available in Both PC-BSD and Solaris

System V Compatible UNIX	BSD Compatible UNIX	Purpose
`lp`	`lpr`	Submits a file for printing
`lpstat`	`lpq`	Shows the status of print jobs for one or more printers
`cancel`	`lprm`	Removes/purges one or more jobs from the print queue
	`lpc`	Activates the printer control program
	`lptest`	Generates ripple pattern for testing the printer

is very important to note that you should never try printing nontext files with the `lp` or `lpr` command, especially files with control characters (e.g., executable files such as **a.out**). Doing so will not print what you want printed and will waste many printer pages. Do not even try testing it. If by accident you do send a print request for a nontext file, turn off the printer immediately and alert your system administrator that you need assistance.

The following is a brief description of the `lp` command.

SYNTAX

`lp [options] file-list`

Purpose: Submit a request to print the files in **file-list**

Commonly used options/features:

`-P page-list`	Print the pages specified in **page-list**
`-d ptr`	Send the print request to the **ptr** printer
`-m`	Send e-mail after printing is complete
`-n N`	Print `N` copies of the file(s) in **file-list**; default is one copy
`-t title`	Print **title** on a banner page
`-w`	Write to user's terminal after printing is complete

The following session shows how to use the `lp` command with and without options. The first command prints the **sample** file on the default printer. The system administrator sets the default printer for the users on a system. The job ID for the first print request is cpr-981, which tells you that the name of the printer is **cpr**. The second command uses the `-d` option to specify that the **sample** file should be printed on the **spr** printer. The third command is for printing three copies each of the **sample** and **phones** files on the **qpr** printer.

```
% lp sample
request id is cpr-981 (1 file(s))
% lp -d spr sample
request id is spr-983 (1 file(s))
% lp -d qpr -n 3 sample phones
request id is qpr-984 (2 file(s))
%
```

As mentioned before, the BSD counterpart of the `lp` command is the `lpr` command. The following is a brief description of this command.

SYNTAX

```
lpr [options] file-list
```

Purpose: Submit a request to print the files in **file-list**

Commonly used options/features:

`-# N`	Print `N` copies of the file(s) in **file-list**; default is one copy
`-P ptr`	Send the print request to the **ptr** printer
`-T title`	Print **`title`** on a banner page
`-m`	Send e-mail after printing is complete
`-p`	Format output by using the `pr` command

The following session shows the BSD versions of the commands that perform the same print tasks as the `lp` command. Thus, the first `lpr` command sends the print request for printing the **sample** file on the default printer. The second command sends the request for printing the **sample** file on the **spr** printer. The third command prints three copies of the **sample** and **phones** files on the **qpr** printer.

```
% lpr sample
% lpr -P spr sample
% lpr -P qpr -# 3 sample
%
```

You can use the following command to print the **sample** file with the header information on every page produced by the `pr` command. The vertical bar (|) is called the *pipe* symbol, which we discuss in detail in Chapter 9.

```
% pr sample | lpr
%
```

You can perform the same task with the `lpr -p sample` command. You can print the **sample** file with line numbers and a `pr` header on each page by using the following command. You can also perform the same task with the `nl sample | lpr -p` command.

```
$ nl sample | pr | lpr
$
```

You can enable the `lpr` command to print a nonstandard text file, such as a TEX file, by specifying an appropriate flag. For example, you can use the `-t` option to print a troff file and the `-n` option to print an nroff file.

6.8.3 Finding the Status of Your Print Requests

In a System V–compatible system, the `lpstat` command can be used to display the status of print jobs on a printer. The following is a brief description of the `lpstat` command.

SYNTAX

`lpstat [options]`

Purpose: Display the status of print jobs on a printer

Commonly used options/features:

`-d`	Display the status of print jobs sent to the default printer using the `lp` command
`-o job-ID-list`	Display the status of the print jobs in **job-ID-list**; separate job IDs with spaces and enclose the requests in double quotes for more than one job
`-p printer-list`	Display the status of print jobs on the printers specified in **printer-list**
`-u user-list`	Display the status of print jobs for the users in **user-list**

Without any option, the `lpstat` command displays the status of all your print jobs that are printing or waiting in the print queue of the default printer. The commands in the following session show some typical uses of the command. The `lpstat -p` command shows the status of all printers on the network. The `lpstat -p qpr` displays the status of print jobs on the **qpr** printer. The `lpstat -u sarwar` displays all print jobs for the user **sarwar**. The output of the command shows that there are three print jobs that **sarwar** has submitted: two to **qpr** (job IDs qpr-3998 and qpr-3999) and one to **tpr** (job ID tpr-203). Finally, the `lpstat -a` command displays all the printers that are up and accepting print jobs.

```
$ lpstat -p
printer cpr is idle. enabled since Tue Sep 2 10:43:48 GMT 2014.
available. printer mpr faulted. enabled since Mon Sep 1 10:48:29
GMT 2014. available. printer qpr now printing qpr-53. enabled
since Mon Sep 1 10:48:29 GMT 2014. available. printer spr is idle.
enabled since Mon Sep 1 10:48:29 GMT 2003. available.
$ lpstat -p qpr
printer qpr now printing qpr-53. enabled since Mon Sep 1 10:48:29
GMT 2014. available.
$ lpstat -u sarwar
qpr-3998 sarwar  93874  Sep    2   22:05 on qpr
qpr-3999 sarwar  93874  Sep    2   22:05
tpr-203  sarwar  93874  Sep    2   22:05 on tpr
$ lpstat -a
cpr accepting requests since Tue Sep 2 10:43:48 GMT 2014
spr accepting requests since Mon Sep 1 10:48:29 GMT 2014
$
```

The following is a brief description of the BSD counterpart of the `lpstat` command, the `lpq` command.

SYNTAX

`lpq [options]`

Purpose: Display the status of print jobs on a printer

Commonly used options/features:

`-P printer-list`	Display the status of print jobs on the printers specified in **printer-list**
`-l`	Display the long format status of print jobs sent using the `lpr` command on the default printer

The most commonly used option is `-P`. In the following session, the first command is used to display the status of print jobs on the **mpr** printer. The output of the command shows that four jobs are in the printer queue: jobs 3991, 3992, 3993, and 3994. The active job is at the head of print queue. When the printer is ready for printing, it will print the active job first. The second command shows that the **qpr** printer does not have any jobs to print.

```
$ lpq -Pmpr
mpr is ready and printing
Rank        Owner       Job   Files              Total Size
active      sarwar      3991  mail.bob           1056 bytes
1st         sarwar      3992  csh.man            93874 bytes
2nd         davis       3993  proposal1.nsa      2708921 bytes
3rd         tom         3994  memo               8920 bytes
$ lpq -Pqpr
no entries
$
```

6.8.4 Canceling Your Print Jobs

If you realize that you have submitted the wrong file(s) for printing, you will want to cancel your print request(s). The System V command for performing this task is `cancel`. The following is a brief description of the command.

SYNTAX

`cancel [options] [printer]`

Purpose: Cancel the print requests sent through the `lp` command—that is, take these jobs out of the print queue

Commonly used options/features:

`-jobID-list`	Cancel the print jobs specified in **jobID-lsit**
`-ulogin`	Cancel all print requests issued by the user **login**

The following commands show how to cancel a print job. You can display the job IDs of the print jobs on a printer by using the `lpstat` or `lpq` command, as shown is Section 6.8.3. The first command cancels the print job mpr-3991. The second command cancels all print requests by the user **sarwar** on all printers. You can cancel your own print jobs only. The last command, therefore, works only when run by **sarwar** or the superuser.

```
$ cancel mpr-3991
request "mpr-3991" canceled
$ cancel -u sarwar mpr
request "mpr-3992" canceled
request "mpr-3995" canceled
$
```

The BSD counterpart of the `cancel` command is `lprm`. The following is a brief description of the command.

SYNTAX

`lprm [options] [jobID-list] [user(s)]`

Purpose: Cancel the print requests made by using the `lpr` command—i.e., remove these jobs from the print queue; the jobIDs in **jobID-list** are taken from the output of the `lpq` command

Commonly used options/features:

`-`	Remove all the print jobs owned by **user**
`-P ptr`	Specify the print queue for the **ptr** printer

The following `lprm` commands perform the same tasks as the `cancel` commands described in the previous session.

```
$ lprm -Pmpr 3991
mpr-3996 dequeued
$ lprm -Pmpr sarwar
mpr-3997 dequeued
mpr-3998 dequeued
$
```

When run without an argument, the `lprm` command removes the job that is currently active, provided it is one of your jobs.

The following in-chapter exercises will give you practice on using the printing-related commands.

EXERCISE 6.11

How would you print five copies of the file **memo** on the printer **ece_hp1**? Give commands for both System V and BSD UNIX.

EXERCISE 6.12

After submitting the two requests, you realize that you really wanted to print five copies of the file **letter**. How would you remove the print jobs from the print queue? Again, give commands for both System V and BSD UNIX.

SUMMARY

The basic file operations involve displaying all or part of a file's contents, renaming a file, moving a file to another file, removing a file, determining a file's size, comparing files, combining files and storing them in another file, appending new contents (which can come from another disk file, keyboard, or output of a command) at the end of a file, and printing files. UNIX provides several commands that can be used to perform these operations.

The `cat` and `more` commands can be used to display all the contents of a file on the display screen. The > symbol can be used to send outputs of these commands to other files, and the >> operator can be used to append new contents at the end of a file. The `cat` command sends a file's contents as continuous text, whereas the `more` command sends them in the form of pages. Furthermore, the `more` command has several useful features, such as the ability to display a page that contains a particular string. The `less` command supports even more features than the `more` command, including the vi-style forward and backward searching.

The `head` and `tail` commands can be used to display the initial or end portions (head or tail) of a file. These helpful commands are usually used to find out the type of data contained in a file, without using the `file` command (see Chapter 4). In addition, the `file` command cannot decipher contents of all the files.

A copy of a file can be made in another file or directory by using the `cp` command. Along with the > operator, the `cat` command can also be used to make a file copy, although there are differences between using the `cp` and `cat >` commands for copying files (see Chapter 9). A file can be moved to another file by using the `mv` command. However, depending on whether the source and destination files are on the same file system, its use might or might not result in actual movement of file data from one location to another. If the source and destination files are on the same file system, the file data is not moved and the source file is simply linked to the new place (destination) through a hard link (see Chapter 8) and the original/source link is removed. If the two files are on different file systems, an actual copy of the source file is made at the new location and the source file is removed (unlinked) from the current directory. Files can be removed from a file structure by using the `rm` command. This command can also be used to remove directories recursively.

The size of a file can be determined by using the `ls -l` or `wc` commands; both give file sizes in bytes. In addition, the `wc` command gives the number of lines and words in the file. Both commands can be used to display the sizes of multiple files by using the shell metacharacters `*`, `?`, `[]`, and `^`.

The `diff` command can be used to display the differences between two files. The command, in addition to displaying the differences between the files, displays useful information in the form of a sequence of commands for the `ed` editor that can be used to make the two files the same. The `uniq` command can be used to remove all but one occurrence of successive repeated lines. With the `-d` option, the command can be used to display the repeated lines.

The `lp` (System V) or `lpr` (BSD) command can be used for printing files on a printer. The `lpstat` (System V) or `lpq` (BSD) commands can be used for checking the status of all print jobs (requests) on a printer (waiting, printing, etc.). The `cancel` (System V) or `lprm` (BSD) commands can be used to remove a print job from a printer queue so that the requested file is not printed. All of these print commands are available in both PC-BSD and Solaris.

QUESTIONS AND PROBLEMS

1. List 10 operations that you can perform on UNIX files.
2. Give a command line for viewing the sizes (in lines and bytes) of all the files in your present working directory.
3. What does the `tail -10r ../letter.John` command do?
4. Give a command for viewing the size of your home directory. Give a command for displaying the sizes of all the files in your home directory.
5. Give a command for displaying all the lines in the **students** file, starting with line 25.
6. Give a command for copying all the files and directories under a directory **courses** in your home directory. Assume that you are in your home directory. Give another command to accomplish the same task, assuming that you are not in your home directory.
7. Repeat Problem 6, but give the command that preserves the modification times and permissions for the file.
8. What is the difference between the `cp –r ~/courses ~/backups` and `cp –r ~/courses/ ~/backups` commands?
9. Give an option for the `rm` command that could protect you from accidently removing a file, especially when you are using wild cards such as `*` and `?` in the command.
10. What do the following commands do?
    a. `cp -f sample sample.bak`
    b. `cp -fp sample sample.bak`
    c. `rm -i ~/personal/memo*.doc`
    d. `rm -i ~/unixbook/final/ch??.prn`
    e. `rm -f ~/unixbook/final/*.o`

f. `rm -f ~/courses/ece446/lab[1-6].[cC]`

g. `rm -r ~/NotNeededDirectory`

h. `rm -rf ~/NotNeededDirectory`

i. `rm -ri ~/NotNeededDirectory`

11. Give a command for moving files **lab1**, **lab2**, and **lab3** from the **~/courses/ece345** directory to a **newlabs.ece345** directory in your home directory. If a file already exists in the destination directory, the command should prompt the user for confirmation.

12. Give a command to display the lines in the **~/personal/phones** file that are not repeated.

13. Refer to In-Chapter Exercise 9. Give a sequence of commands to save the sequence of commands for the `ed` editor and use them to make **sample** and **example** the same files.

14. You have a file in your home directory called **tryit&**. Rename this file. What command did you use?

15. Give a command for displaying attributes of all the files starting with a string **prog**, followed by zero or more characters and ending with a string **.c** in the **courses/ece345** directory in your home directory.

16. Refer to Problem 15. Give a command for file names with two English letters between **prog** and **.c**. Can you give another command line to accomplish the same task?

17. Give a command for displaying files **got|cha** and **M*A*S*H** one screenful at a time.

18. Give a command for displaying the sizes of files that have the **.jpg** extension and names ending with a digit.

19. What does the `rm *[a-zA-Z]??[1,5,8].[^p]*` command do?

20. Give a command to compare the files **sample** and **example** in your present working directory. The output should generate a series of commands for the `ed` editor.

21. Give a command for producing 10 copies of the report **file** on the **ece_hp3** printer. Each page should contain a page header produced by the `pr` command. Give commands for both System V and BSD UNIX.

22. Give the command to print the nroff file **Chapter 1** by using the `lpr` command. What command line would you use to print the troff file **sample** with the `lpr` command?

23. Give a command for checking the status of a print job with job ID ece_hp3-8971. Is this command for System V or BSD? How would you remove this print job from the print queue? Give commands for both System V and BSD UNIX.

24. What is the difference between the `tail -15 file1` and `tail +15 file1` commands? Which of the following commands is equivalent to cat file1: `tail -$ file`, `tail -1 file1`, or `tail +1 file1`? Why?
25. What is the purpose of the `more –n5 file1`?
26. What are the differences between the `more` and `less` commands? Which of the two is more powerful and user friendly in your opinion and why?
27. Create a file in your current directory called **f1**. What is the inode number of the file? Move the file to your home directory and name it **f1.moved**. What is the inode number of the moved file? Move the moved file to the **/tmp** directory. What is the inode number of the **/tmp/f1.moved** file? Why are the inode numbers for **f1** and **f2** the same? Why is the inode number of **/tmp/f1.moved** different from **f1** (or ~**/f1.moved**)?

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

CHAPTER 7

Advanced File Processing

Objectives

- To explain file compression and how it can be performed
- To explain the sorting process and how files can be sorted
- To discuss searching for commands and files in the UNIX file structure
- To discuss the formation and use of regular expressions
- To describe searching files for expressions, strings, and patterns
- To describe how database-type operations of cutting and pasting fields in a file can be performed
- To discuss encoding and decoding of files
- To explain file encryption and decryption
- To cover the commands and primitives

```
>, ~, compress, crypt, crypto, cut, egrep, fgrep, find, grep,
openssl, paste, pcat, sort, uncompress, uuencode, uudecode,
whereis, which, zcat
```

7.1 INTRODUCTION

In this chapter, we describe some of the more advanced file-processing operations and show how they can be performed in UNIX. But before describing these operations, we discuss the important topic of *regular expressions*, which are a set of rules that can be used to specify one or more strings using a sequence of special text characters. While discussing the operations, we also describe the related shell commands and tools that make use of regular expressions. We also give examples to illustrate how these commands can be used to perform the required operations.

7.2 COMPRESSING FILES

Reduction in the size of a file is known as file compression, which has both space and time advantages. A compressed file takes less disk space to store, less time to transmit from one computer to another in a network or internet environment, and less time to copy. It takes time to compress a file, but if a file is to be copied or transmitted several times, the time spent compressing the file could be just a fraction of the total time saved. In addition, if the compressed file is to be stored on a secondary storage device (e.g., a disk) for a long time, the savings in disk space can be considerable. Another consequence of compression is that a compressed file reads as garbage. However, this is not a problem because the process is fully reversible and a compressed file can be converted back to its original form. If you can fully recover the original file from its compressed version, the compression is known as *lossless compression*. Lossless compression techniques are normally used for text files. There are *lossy compression* techniques too. These techniques are used to reduce the size of a file for storing, handling, and transmitting its content between computers on a network. Lossy compression techniques are normally used for image files such as JPEGs. In this chapter, we discuss tools for compressing and decompressing text and executable files. We also discuss commands for displaying and searching compressed files.

The UNIX operating system has many commands for compressing and decompressing files and for performing various operations on compressed files. These commands include the traditional UNIX commands for compressing and decompressing files, `compress` and `uncompress`, and the GNU tools `gzexe` (compress executable files), `gzip` (for compressing files), `gunzip` (for uncompressing files that were compressed with `gzip`), `zcat` (for displaying compressed files; `gzcat` does the same), `gzcmp` (for comparing compressed files), `gzforce` (for forcing the **.gz** extension onto compressed files so that `gzip` will not compress them twice), `gzmore` (for displaying compressed files one page at a time), and `gzgrep` (the `grep` command for compressed files; it searches possibly compressed files for a regular expression). This section will primarily discuss file compression and decompression. Although the GNU tools (`gzip` and `gunzip`) are better than `compress` and `uncompress` commands, we discuss both sets of commands for completion.

7.2.1 The `compress` Command

The `compress` command reads contents of files that are passed to it as parameters, analyzes their contents for repeated patterns, and then substitutes a smaller number of characters for these patterns by using adaptive Lempel–Ziv coding. A compressed file's contents are altogether different from the original file. The compressed file contains nonprintable characters, so displaying a compressed file on the screen shows a bunch of control characters, or garbage. The `compress` command saves the compressed file in a file that has the same name as the original file, with an extension **.Z** appended to it. The file has the same access permissions and modification date as the original file. The original file is removed from the file structure.

The following is a brief description of the `compress` command; the syntax of and options used in the `uncompress` command are exactly the same.

SYNTAX

`compress [option] file-list`

Purpose: Compress files in **file-list**
Output: The compressed **.Z** file or standard output if input is from standard input
Commonly used options/features:
- **`-c`** Write compressed file to the display screen instead of a **.Z** file
- **`-f`** Force compression; no prompts
- **`-v`** Display compression percentage and the names of compressed files

With no file argument or `-` as an argument, the `compress` command takes input from standard input (keyboard by default), which allows you to use the command in a pipeline (see Chapter 9). We normally use the command with one or more files as its arguments. In the following session, we use the command with one file, **t2**, as its argument. The compressed file is stored in the **t2.Z** file, and the **t2** file is removed from the file system.

```
$ cat t2
This file is being used to test various commands and tools. Long
live UNIX! UNIX rules the networking world!!
$ compress t2
$ cat t2.Z
¨6e@
      !_L¨7g@ÔS
            :o.C
                  0rÒ_¨'ÌÂ¸!CP¥EoØÌq7A°Ic'a'I°
                        P_3)uÎ 1
$
```

The following session illustrates the use of `compress` with the `-v` (verbose) option and multiple files. Note that the use of the shell metacharacter `?` denotes a single character and that the output of the `compress` command with the `-v` option shows the percentage of compression performed on the source files. Note that **t2** and **t3** have been compressed only a little, but **t4** has been compressed to 58% of its original size, from 4083 bytes to 2371 bytes.

```
% ls -l t?
-rw-r--r--  1 sarwar  faculty   116 Aug 30 19:35 t2
-rw-r--r--  1 sarwar  faculty   99 Aug 30 19:32 t3
-rw-r--r--  1 sarwar  faculty   4083 Aug 30 19:33 t4
% compress -v t2 t3 t4
t2.Z: 93% compression
t3.Z: 90% compression
t4.Z: 58% compression
% ls -l t?.Z
```

```
-rw-r--r--  1 sarwar  faculty   108 Aug 30 19:35 t2.Z
-rw-r--r--  1 sarwar  faculty   89 Aug 30 19:32 t3.Z
-rw-r--r--  1 sarwar  faculty   2371 Aug 30 19:33 t4.Z
%
```

7.2.2 The uncompress Command

You can use the `uncompress` command to uncompress the compressed files and put them in the corresponding original files (i.e., the names of the compressed files without **.Z** extensions). The following session shows the use of the `uncompress` command. Note that `uncompress` ignores the `–v` option.

```
% uncompress t1.Z t2.Z t3.Z t4.Z
% ls -l t?
-rw-r--r--  1 sarwar  faculty   116 Aug 30 19:35 t2
-rw-r--r--  1 sarwar  faculty   99 Aug 30 19:32 t3
-rw-r--r--  1 sarwar  faculty   4083 Aug 30 19:33 t4
%
```

The `uncompress -v t?.Z` command performs the same task if the current directory contains only these four file names that start with **t** followed by one other character and have the **.Z** extension. Note that the output of the `uncompress` command goes into a file with the original file name.

As given in the command description, multiple files can be displayed by the `zcat` command. For example, the command `zcat t2.Z t3.Z t4.Z` displays the uncompressed forms of the three files **t2.Z**, **t3.Z**, and **t4.Z**.

7.2.3 The gzip Command

The `gzip` command is the GNU tool for compressing files. The compressed file is saved as a file that has the same name as the original, with an extension **.gz** appended to it. As is the case with the `compress` command, the compressed files retain the access/modification times, ownership, and access privileges of the original files. The original file is removed from the file structure. With no file argument or with `-` as an argument, the `gzip` command takes input from standard input (keyboard by default), which allows you to use the command in a pipeline (see Chapter 9). We normally use the command with one or more files as its arguments. Here is a brief description of the command.

SYNTAX

`gzip [option] [file-list]`

Purpose: Compress each file in **file-list** and store it in **filename.gz**, where **filename** is the name of the original file. If no file is specified in the command line or if – is specified, take input from standard input.

Output: The compressed **.gz** file or standard output if input is from standard input

Commonly used options/features:

`-N`	Control compression speed (and compression ratio) according to the value of **N**, with 1 being fastest and 9 being slowest; slow compression compresses more
`-c`	Send output to standard out; input files remain unchanged
`-d`	Uncompress a compressed (**.gz**) file
`-f`	Force compression of a file when its **.gz** version exists, or it has multiple links, or input file is **stdin**
`-l`	For the compressed files given as arguments, display sizes of the uncompressed and compressed versions, compression ratio, and uncompressed name
`-r`	Recursively compress files in the directory specified as arguments
`-t`	Test integrity of the compressed files specified as arguments
`-v`	Display compression percentage and the names of compressed files

7.2.4 The `gunzip` Command

The `gunzip` command can be used to perform the reverse operation and bring compressed files back to their original forms. The `gzip -d` command can also perform this task. With the `gunzip` command, the `-N`, `-c`, `-f`, `-l`, and `-r` options work just like they do with the `gzip` command.

The following session shows the use of the two commands with and without arguments. We use the `man bash > bash.man` and `man tcsh > tcsh.man` commands to save the manual page for the Bourne Again and TC shells in the **bash.man** and **tcsh.man** files, respectively. The `gzip bash.man tcsh.man` command is used to compress the **bash.man** and **tcsh.man** files, and the `gzip –l bash.man.gz tcsh.man.gz` command is used to display some information about the compressed and uncompressed versions of the **bash.man** and **tcsh.man** files. The output of the command shows, among other things, the percentage of compression achieved: 71.3% for **bash.man** and 67.6% for **tcsh.man**. The `gzip bash.man.gz` command is used to show that `gzip` does not compress an already compressed file that has a **.gz** extension. If a compressed file does not have the **.gz** extension, `gzip` will try to compress it again. The `gunzip *.man.gz` command is used to decompress the **bash.man.gz** and **tcsh.man.gz** files. The `gzip –d *.man.gz` command can be used to perform the same task. The `ls –l` commands have been used to show that the modification time, ownership, and access privileges of the original file are retained for the compressed file.

```
% man bash > bash.man
% man tcsh > tcsh.man
% ls -l *.man
-rw-r--r--  1 sarwar  faculty  367470 Aug 30 21:39 bash.man
-rw-r--r--  1 sarwar  faculty  239996 Aug 30 21:39 tcsh.man
% gzip bash.man
% ls -l bash.man.gz
-rw-r--r--  1 sarwar  faculty  105342 Aug 30 21:39 bash.man.gz
% gzip bash.man.gz
gzip: bash.man.gz already has .gz suffix – unchanged
```

```
% gunzip bash.man.gz
% gzip bash.man tcsh.man
% gzip -l bash.man.gz tcsh.man.gz
  compressed uncompressed  ratio uncompressed_name
      105342       367470  71.3% bash.man
       77715       239996  67.6% tcsh.man
      183057       607466  69.8% (totals)
% gunzip *.man.gz
% ls -l *.man
-rw-r--r--  1 sarwar  faculty  367470 Aug 30 21:39 bash.man
-rw-r--r--  1 sarwar  faculty  239996 Aug 30 21:39 tcsh.man
%
```

7.2.5 The gzexe Command

The gzexe command can be used to compress executable files. An executable file compressed with the gzexe command remains executable and can be executed by using its name. This is not the case if an executable file is compressed with the gzip command. Therefore, an executable file is compressed with the gzexe command in order to save disk space and network bandwidth if the file is to be transmitted from one computer to another—for example, via e-mail over the Internet. The following is a brief description of this command.

SYNTAX

gzexe [options] [file-list]

Purpose: Compress the executable files given in **file-list**; backup files are created in **filename~** and should be removed after the compressed files have been successfully created

Commonly used options/features:

-d Decompress compressed files

The following session illustrates the use of the gzexe command. Note that when you compress the executable file **sh.temp** with the gzexe command, it creates a backup of the original file in the **sh.temp**~ file. After the **sh.temp** file has been compressed, it can be executed as an ordinary executable file. The gzexe -d sh.temp command is used to decompress the compressed file **sh.temp**. The backup of the compressed version is saved in the **sh.temp~** file.

```
% cp /bin/sh sh.temp
% file sh.temp
sh.temp: ELF 64-bit LSB executable, x86-64, version 1 (FreeBSD),
dynamically linked (uses shared libs), for FreeBSD 10.0 (1000510),
stripped
% gzexe sh.temp
```

```
sh.temp:       49.2%
% ls -l sh*
-rwx------  1 sarwar  faculty   71133 Aug 30 21:59 sh.temp
-rwx------  1 sarwar  faculty  139264 Aug 30 21:59 sh.temp~
% gzexe -d sh.temp
49.2%
% ls -l sh*
-rwx------  1 sarwar  faculty  139264 Aug 30 22:00 sh.temp
-rwx------  1 sarwar  faculty  139264 Aug 30 21:59 sh.temp~
%
```

7.2.6 The zcat and zmore Commands

Converting the compressed file back to the original and then displaying it is a time-consuming process because file creation requires disk I/O. If you only want to view the contents of the original file, you can use the UNIX command `zcat` (the `cat` command for compressed files), which displays the contents of files compressed with `compress` or `gzip`. The command uncompresses a file before displaying it. The original file remains unchanged. The `zmore` command can be used to display the compressed files one screenful at a time. When no file or `-` is given as a parameter, these commands read input from **stdin**. Both commands allow you to specify one or more files as parameters. Here is a brief description of the `zcat` command.

SYNTAX

`zcat [options] [file-list]`

Purpose: Concatenate compressed files in their original form and send them to standard output; if no file is specified, take input from standard input

Commonly used options/features:

`-h` Display help information
`-r` Operate recursively on subdirectories
`-t` Test integrity of compressed files

In the following session, the `gzip` command is used to compress the **bash.man** file and store it in the **bash.man.gz** file. When the `more` command is used to display a compressed file, garbage is displayed on the screen. The `zmore` command is used to display the contents of the original file. We did not use the `zcat` command because **bash.man** is a large, multipage file.

```
$ gzip bash.man
$ more bash.man.gz
<8B>^H^HA<FE>T<97><C3>
<DD><A0><CC>□<AE>j<82><B0>G<FE><EC>o<DC><99>U<DD><8F><AC><"<E3>
<8E>_ƒ<FA><E4><D9>o<8F><F9>O<E4><FF>ƒų<BE><FF><F9><BA><FA><FA>I
```

```
<F5><E4>w<D5>□<BE>xO<DE>><AF><FB><AB>x<BF><FA><FA><BB><F8>i<B7>□
<9B><93>'<97>u<BB><8E><CF>'<DB>,<97><EF><BD><F7><AC>z<F6><A7><EA>O
<F0><C5>o<AA>o<BE><AD><BE><85>?<9F>VO<E1>_<FB>zu^Wu_<F5>W<D5>Uƒ
<D1>m<86><B6>ÿ<F7>/<E3><8B>y<B7>Z<D5><EB><C5>Y?l<DB><F5>e<CF>
p<D1>. <9B><97><EF><BD><F7>Y<F5><8F><A3><FE><BE><FA>=
...
% zmore bash.man.gz
BASH(1)                                                     BASH(1)
NAME
       bash - GNU Bourne-Again SHell
SYNOPSIS
       bash ÿoptions¦ ÿcommand_string ƒ file¦
COPYRIGHT
       Bash is Copyright (C) 1989-2013 by the Free Software
Foundation, Inc.
DESCRIPTION
       Bash  is  an  sh-compatible  command language interpreter
that executes commands read from the standard input or from a
file.  Bash also incor-
...
%
```

In the following session, the zcat command decompresses the **t2.gz** file and sends its output to standard output (the display screen in this case). The file **t2.gz** remains intact.

```
% gzip t2
% zcat t2.gz
This file will be used to test various UNIX and Linux commands and
tools.
UNIX and Linux rule the networking world!
%
```

As given in this command description, multiple files can be displayed by the zcat command. For example, `zcat  t1.Z  t2.Z  t3.Z` may be used to display the uncompressed forms of the three files **t1.Z**, **t2.Z**, and **t3.Z**. The zcat command may also be used to display the files compressed with the compress command—that is, files with the **.Z** extension.

In the following in-chapter exercises, you will use the compress, uncompress, gzip, gunzip, gzmore, and zcat commands to appreciate their syntax and semantics.

EXERCISE 7.1

Create the **t2** file used in this section. Use the compress command to compress the file. What command line did you use?

EXERCISE 7.2

Create the **bash.man** file used in this section. Use the `gzip` command to compress the file. What command line did you use?

EXERCISE 7.3

Display the compressed version of the **t2** file on the display screen. What command line did you use?

EXERCISE 7.4

Give the command line for uncompressing the compressed files generated in Exercises 7.1 and 7.2. Where does the uncompressed (original) file go? Also, repeat the shell sessions shown in Sections 7.2.1 through 7.2.6.

7.3 SORTING FILES

Sorting means ordering a set of items according to some criteria. In computer jargon, it means ordering a set of items (e.g., integers, a character, or strings) in *ascending* (the next item is greater than or equal to the current item) or *descending* (the next item is less than or equal to the current item) order. So, for example, a set of integers {10, 103, 75, 22, 97, 52, 1} would become {1, 10, 22, 52, 75, 97, 103} if sorted in ascending order, and {103, 97, 75, 52, 22, 10, 1} if sorted in descending order. Similarly, words in a dictionary are listed in ascending order. Thus, the word `apple` appears before the word `apply`.

Sorting is a commonly used operation and is also performed in a variety of software systems. Systems in which sorting is used include

- Words in a dictionary
- Names of people in a telephone directory
- Airline reservation systems that display arrival and departure times for flights sorted according to flight numbers at airport terminals
- Names of people displayed in a pharmacy with ready prescriptions
- Names of students listed in class lists coming from the registrar's office

The sorting process is based on using a field, or portion of each item, known as the sort key. In order to determine the position of each item in the sorted list, you compare the items in a list (usually two at a time) by using their key fields. The choice of the field used as the key depends on the items to be sorted. If the items are personal records (e.g., student employee records), last name, student ID, and social security number are some of the commonly used keys. If the items are arrival and departure times for the flights at an airport, flight number and city name are commonly used keys.

The UNIX `sort` utility can be used to sort items in text (ASCII) files. The following is a brief description of this utility.

SYNTAX

`sort [options] [file-list]`

Purpose: Sort lines in the ASCII files in **file-list**
Output: Sorted files to standard output
Commonly used options/features:

`-b`	Ignore leading blanks
`-d`	Sort according to usual alphabetical order: ignore all characters except letters, digits, and then blanks
`-f`	Consider lowercase and uppercase letters to be equivalent
`+n1[-n2]`	Specify a field as the sort key, starting with **`+n1`** and ending at **`-n2`** (or end of line if **`-n2`** is not specified); field numbers start with 0
`-r`	Sort in reverse order

If no file is specified in **file-list**, `sort` takes input from standard input. The output of the `sort` command goes to standard output. By default, `sort` takes each line, starting with the first column, to be the key. In other words, it rearranges the lines of the file—that is, strings separated by the newline character (n)—according to the contents of all the fields, going from left to right. The following session illustrates the use of `sort` with and without some options. The **students** file contains the items (student records, one per line) to be sorted. Each line contains four fields: first name, last name, e-mail address, and phone number. Each field is separated from the next by one or more space characters.

```
% cat students
John Johnsen        john.johnsen@tp.com        503.555.1111
Hassaan Sarwar      hsarwar@k12.st.or          503.444.2132
David Kendall       d_kendall@msnbc.org        229.111.2013
John Johnsen        j.johnsen@psu.net          301.999.8888
Ibraheem Sarwar     ibraheem@abc.sci.com       222.123.4567
Kelly Kimberly      kellyk@umich.gov           555.123.9999
Maham Sarwar        smsarwar@k12.st.or         713.888.0000
Jamie Davidson      j.davidson@uet.edu         515.001.1212
Nabeel Sarwar       n.sarwar@xyz.net           434.555.1212
% sort students
David Kendall       d_kendall@msnbc.org        229.111.2013
Hassaan Sarwar      hsarwar@k12.st.or          503.444.2132
Ibraheem Sarwar     ibraheem@abc.sci.com       222.123.4567
Jamie Davidson      j.davidson@uet.edu         515.001.1212
John Johnsen        j.johnsen@psu.net          301.999.8888
John Johnsen        john.johnsen@tp.com        503.555.1111
Kelly Kimberly      kellyk@umich.gov           555.123.9999
Maham Sarwar        smsarwar@k12.st.or         713.888.0000
```

```
Nabeel Sarwar       n.sarwar@xyz.net            434.555.1212
%
```

Note that the lines in the **students** file are sorted in ascending order by all characters, going from left to right (the whole line is used as the sort key). The following command sorts the file by using the whole line, starting with the last name—the second field (field number 1)—as the sort key.

```
% sort +1 students
Jamie Davidson      j.davidson@uet.edu      515.001.1212
John Johnsen        j.johnsen@psu.net       301.999.8888
John Johnsen        john.johnsen@tp.com     503.555.1111
David Kendall       d_kendall@msnbc.org     229.111.2013
Kelly Kimberly      kellyk@umich.gov        555.123.9999
Hassaan Sarwar      hsarwar@k12.st.or       503.444.2132
Ibraheem Sarwar     ibraheem@abc.sci.com    222.123.4567
Nabeel Sarwar       n.sarwar@xyz.net        434.555.1212
Maham Sarwar        smsarwar@k12.st.or      713.888.0000
%
```

The following command sorts the file in reverse order by using the phone number as the sort key and ignoring leading blanks (spaces and tabs). The +3 option specifies the phone number to be the sort key (as phone number is the last field), the -r option informs sort to display the sorted output in reverse order, and the -b option asks the sort utility to ignore the leading white spaces between fields.

```
% sort +3 -r -b students
Maham Sarwar        smsarwar@k12.st.or      713.888.0000
Kelly Kimberly      kellyk@umich.gov        555.123.9999
Jamie Davidson      j.davidson@uet.edu      515.001.1212
John Johnsen        john.johnsen@tp.com     503.555.1111
Hassaan Sarwar      hsarwar@k12.st.or       503.444.2132
Nabeel Sarwar       n.sarwar@xyz.net        434.555.1212
John Johnsen        j.johnsen@psu.net       301.999.8888
David Kendall       d_kendall@msnbc.org     229.111.2013
Ibraheem Sarwar     ibraheem@abc.sci.com    222.123.4567
%
```

The -b option is important if fields are separated by more than one space and the number of spaces differs from line to line, as is the case for the **students** file. The reason is that the space character is "smaller" (in terms of its ASCII value) than all letters and digits, and not skipping blanks will generate unexpected output. The sort keys can be combined, with one being the primary key and others being secondary keys, by specifying them in the order of preferences (the primary key occurring first). The following command sorts the **students** file with the last name as the primary key and the phone number as the secondary key.

```
% sort +1 -2 +3 -b students
Jamie Davidson      j.davidson@uet.edu       515.001.1212
John Johnsen        j.johnsen@psu.net        301.999.8888
John Johnsen        john.johnsen@tp.com      503.555.1111
David Kendall       d_kendall@msnbc.org      229.111.2013
Kelly Kimberly      kellyk@umich.gov         555.123.9999
Ibraheem Sarwar     ibraheem@abc.sci.com     222.123.4567
Nabeel Sarwar       n.sarwar@xyz.net         434.555.1212
Hassaan Sarwar      hsarwar@k12.st.or        503.444.2132
Maham Sarwar        smsarwar@k12.st.or       713.888.0000
%
```

The primary key is specified as `+1 -2`, meaning that the key starts with the last name (`+1`) and ends before the e-mail address field (`-2`) starts. The secondary key starts at the phone number field (`+3`) and ends at the end of line. As no field follows the phone number, it alone comprises the secondary key. For our file, however, the end result will be the same as for the command `sort +1 students` because the first John Johnsen's e-mail address is "smaller" than the second's.

Exercise 7.5

Repeat the above sessions on your system and verify that the sort command works on your system as expected.

7.4 SEARCHING FOR COMMANDS AND FILES

At times, you will need to find whether a particular command or file exists in your file structure. Or, if you have multiple versions of a command, you might want to find out which one executes when you run the command. We discuss three commands that can be used for this purpose: `find`, `whereis`, and `which`.

You can use the `find` command to search a list of directories that meet the criteria described by the expression (see the command description) passed to it as an argument. The command searches the list of directories recursively; that is, all subdirectories at all levels under the list of directories are searched. The following is a brief description of the command.

SYNTAX

`find directory-list expression`

Purpose: Search the directories in **directory-list** to locate files that meet the "criteria" described by the **`expression`** (the second argument); the expression comprises one or more "criteria" (see the examples)

Output: None unless it is explicitly requested in **`expression`**

Commonly used options/features:

`-exec CMD`	The file being searched meets the criteria if the command `CMD` returns 0 as its exit status (true value for commands that execute successfully); `CMD` must terminate with a quoted semicolon (`\;`)
`-inum N`	Search for files with inode number **`N`**
`-links N`	Search for files with `N` links
`-name pattern`	Search for files that are specified by the **`pattern`**
`-newer file`	Search for files that were modified after **file** (i.e., are newer than **file**)
`-ok CMD`	Like **`-exec`** except that the user is prompted first
`-perm octal`	Search for files if permission of the file is **`octal`**
`-print`	Display the pathnames of the files found by using the rest of the criteria
`-size ±N[c]`	Search for files of size `N` blocks; `N` followed by **`c`** can be used to measure size in characters; **`+N`** means size > `N` blocks, and `–N` means size < `N` blocks
`-user name`	Search for files owned by the user name or ID **`name`**
`\( expr \)`	True if **`expr`** is true; used for grouping criteria combined with OR or **`AND`**
`! expr`	True if **`expr`** is false

More criteria are presented in Appendix A. You can use `[-a]` or a space to logically AND, and `-o` to logically OR two criteria. Note that at least one space is needed before and after an open bracket (`[`) or a close bracket (`]`), and before and after `-o`. A complex expression can be enclosed in parentheses, `\(` and `\)`. We now discuss some illustrative examples.

The most common use of the `find` command is to search one or more directories for a file, as shown in the first example. Here, the command searches for the **USA.gif** and **Pakistan.gif** files in your home directory and displays the pathname of the directory that contains them. If the file(s) being searched for occurs in multiple directories, the pathnames of all the directories are displayed.

```
$ find ~ \( -name USA.gif -o -name Pakistan.gif \) -print
/home/sarwar/myweb/USA.html
/home/sarwar/myweb/Pakistan.html
$
```

The following command displays the absolute pathnames of all the files in your home directory that end in **.c** and **.C**.

```
$ find ~ \( -name '*.c' –o –name '*.C' \) –print
...
$
```

The next command searches the **/usr/include** directory recursively for a file named **socket.h** and prints the absolute pathname of the file.

```
$ find /usr/include -name socket.h -print
/usr/include/sys/socket.h
$
```

You might want to know the pathnames for all the hard links (discussed in Chapter 8) to a file—that is, files that have the same inode number. The following command recursively searches the **/usr** and **.** (the present working directory) directories for all the files that have inode number 1749 and prints the absolute pathnames of all such files.

```
$ find /usr/include . -inum 1749 -print
/usr/include/sys/file.h
$
```

The following command searches the present working directory for files that have the name **core** or have extensions **.ps** or **.o**, displays their absolute pathnames, and removes them from the file structure. Parentheses are used to enclose a complex criterion. Be sure that you use spaces before and after `\(`, `\)`, and `-o`. The command does not prompt you for permission to remove; in order to be prompted, replace `-exec` with `-ok`.

```
$ find . \( -name core -o -name '*.ps' -o -name '*.o' \) -print
-exec rm {} \;
...
$
```

You can use the `whereis` command to find out whether your system has a particular command, and if it does, where it is in the file structure. You typically need to get such information when you are trying to execute a command that you know is valid but that your shell cannot locate because the directory containing the executable for the command is not in your search path (see Chapters 2 and 4). Under these circumstances, you can use the `whereis` command to find the location of the command and update your search path. Although `whereis` is a BSD command, most UNIX systems today have it because they have a BSD compatibility package. Depending on the system you are using, the command not only gives you the absolute pathname for the command that you are searching for, but it also gives you the absolute pathnames for its manual page and source files if they are available on your system. The following is a brief description of the command.

SYNTAX

```
whereis [options] [file-list]
```

Purpose: Locate binaries (executable files), source codes, and manual pages for the commands in **file-list—**a space-separated list of command names

Output: Absolute pathnames for the files containing binaries, source codes, and manual pages for the commands in **file-list**

Commonly used options/features:

-b	Search for binaries (executable files) only
-s	Search for source code only

The following examples illustrate use of `whereis` command. The first command is used to locate the `ftp` command. The second command is used to locate the executable file for the `cat` command. The last command locates the information for the `find`, `compress`, and `tar` commands.

```
$ whereis ftp
ftp: /usr/bin/ftp /usr/share/man/en.UTF-8/man1/ftp.1.gz
$ whereis -b cat
cat: /bin/cat
$ whereis find compress tar
find: /usr/bin/find /usr/share/man/en.UTF-8/man1/find.1.gz
compress: /usr/bin/compress /usr/share/man/en.UTF-8/man1/
compress.1.gz
tar: /usr/bin/tar /usr/share/man/en.UTF-8/man1/tar.1.gz
$
```

In the outputs of these commands, the directories **/usr/bin**, **/usr/local/bin**, **/usr/ucb**, and **/usr/sbin** contain the executables for commands, the directory **/usr/man** contains several subdirectories that contain various sections of the UNIX online manual, the file **/etc/tar** is a symbolic link to **/usr/sbin/tar**, and the **/usr/include** directory contains header files.

In a system that has multiple versions of a command, the `which` utility can be used to determine the location (absolute pathname) of the version that is executed by the shell you are using when you type the command. When a command does not work according to its specification, the `which` utility can be used to determine the absolute pathname of the command version that executes. A local version of the command may execute because of the way the search path is set up in the PATH variable (see Chapters 2 and 4). And, the local version has been broken due to a recent update in the code; perhaps it does not work properly with the new libraries that were installed on the system. The `which` command takes **command-list** (actually a **file-list** for the commands) as an argument and returns absolute pathnames for them to standard output.

In the following in-chapter exercises, you will get practice using the `find`, `sort`, and `whereis` commands, as well as appreciate the difference between the `find` and `whereis` commands.

Exercise 7.6

Give a command for sorting a file called **students** by using the whole line starting with the e-mail address.

Exercise 7.7

Give a command for finding out where the executable code for the `traceroute` command is on your system.

Exercise 7.8

You have a file called **phones** somewhere in your directory structure, but you do not remember the pathname of the directory it is in. What command would you use to locate it?

7.5 REGULAR EXPRESSIONS

A *regular expression* is a sequence of constants and operator symbols (known as operators) that represents a set of strings, commonly known as *search patterns*, used for searching file contents for the desired strings. Different tools and commands in UNIX support different sets of operators, but the following operators are supported by almost all UNIX tools that support regular expressions: `()`, `[]`, `.`, `^`, `$`, and `*`. The `*` operator may be used to specify zero or more occurrences of the preceding element. For example, `a*` represents an empty string, `a`, `aa`, `aaa`, and so on. The regular expression `^a` represents a line starting with `a`. Similarly, `.com` represents string `Acom`, `acom`, `Bcom`, `bcom`, `Ccom`, `ccom`, and so on. The regular expression `aa*` represents `a`, `aa`, `aaa`, and so on. It is equivalent to the regular expression `a+`.

Some of the tools use additional operators like |, ?, and +. The `?` operator may be used to specify zero or one occurrence of the preceding element (a character or a pattern). For example, `a?` specifies a string with no characters (an empty string) or a string with only `a`. The + operator may be used to specify one or more occurrence of the preceding element. For example, `a+` represents `a, aa`, `aaa`, and so on. The | specifies alternatives. For example, `a|b`, `pray|prey`, or `a|b*`. The regular expression `a|b*` represents an empty string, `a`, `b`, `bb`, `bbb`, and so on. The `()` operator is used to specify the scope and precedence of operators. For example, `pr(a|e)y` represents `pray` or `prey`, and is thus equivalent to `pray|prey`.

Some of the commonly used UNIX tools that allow the use of regular expressions are `awk`, `ed`, `egrep`, `grep`, `sed`, `vi`, and `vim`, but the level of support for regular expressions isn't the same for all these tools. Whereas `awk` and `egrep` have the best support for regular expressions, `grep` has the weakest.

Table 7.1 lists the regular expression operators, their names, example usage, meanings, and tools that support them. The regular expression operators overlap with shell metacharacters, but you can use single quotes around them to prevent the shell from interpreting them. The word "All" in the last column means that all the tools mentioned support the corresponding operator. We do not use quotes for strings in the fourth column for brevity.

Table 7.2 lists some commonly used regular expressions in the vim editor and their meanings. Needless to say, regular expressions are used in the vim commands. We discuss examples for `grep` and `egrep` in Section 7.6. In the regular expression `/\.c/`, the backslash character (`\`) is used to escape the special meaning of dot (`.`) and take its literal meaning.

Table 7.3 lists some examples of the vim commands that use regular expressions and their meaning. Note that these commands are used when you are in vim's command mode.

In the following in-chapter exercises, you will use regular expressions in the vim editor to appreciate their power.

TABLE 7.1 Regular Expression Operators and Their Support by UNIX Tools

Name/Function	Operator	Example Usage	Meaning	Supported by
Alternation (OR)	|	`x|y|z`	x, y, or z	`awk, grep`
Any character	`.`	`.com`	Acom, acom, Bcom, bcom, Ccom, ccom, …	All
Beginning of line	`^`	`^x`	A line starting with x	All
Concatenation (AND)	`None`	`xyx`	xyz	All
End of line	`$`	`x$`	A line ending with x	All
Escape sequence: Cancels the special meaning of the metacharacter that follows it	`\`	`\*`	*	`ed, sed, vi`
Delimiter: Marks the beginning or end of a regular expression	`/`	`/L..e/`	Love, Live, Lose, Lase, …	`ed, sed, vi`
Grouping	`( ) or \(\)`	`(xy)+`	xy, xyxy, xyxyxy, ...	All
Optional	`?`	`xy?`	x, xy	`awk, egrep`
Repetition (0 or more times)	`*`	`xy*`	x, xy, xyy, xyyy, ...	All
Repetition (1 or more times)	`+`	`xy+`	xy, xyy, xyyy, ...	`awk, egrep`
Matches any character enclosed in brackets. Matches any character not enclosed in brackets.	`[ ]` `[^]`	`/[Hh]ello/` `/[^A-KM-Z]ove/`	Hello, hello Love	All

TABLE 7.2 Examples of Regular Expressions for vim and Their Meaning

Regular Expression	Meaning	Examples
`/^Yes/`	A line starting with the string **Yes**	**Yes**… **Yes**teryear… **Yes**terday… and so on
`/th/`	Occurrence of the string **th** anywhere in a word	**th**e, **th**ere, pa**th**, ba**th**ing, and so on
`/:$/`	A line ending with a colon	… following: … below: … follows: and so on
`/[0-9]/`	A single digit	0, 1, …, 9
`[a-z][0-9]/`	A single lowercase English letter followed by a single digit	a0, a1, ..., a9, …, b0, b1, …, b9, ..., z0, … z1, ..., z9
`/\.c/`	Any word that ends with **.c** (all C source code files)	**lab1.c**, **program1.c**, **client.c**, **server.c**, and so on
`/[a-zA-Z]*/`	Any string composed of letters (uppercase or lowercase) and spaces; no numbers and punctuation marks	All strings without numbers and punctuation marks such as Hello world

TABLE 7.3 Some Commonly Used vim Commands Illustrating the Use of Regular Expressions

`/ [0-9] /`	Do a forward search for a single stand-alone digit character in the current file; digits that are part of strings are not identified.
`?\.c[1-7] ?`	Do a backward search for words or strings in words that end with **.c** followed by a single digit between 1 and 7.
`:1,$s/:$/./`	Search the whole file and substitute a colon (:) at the end of a line with a period (.).
`:.,$s/^[Hh]ello / Greetings /`	From the current line to the end of file, substitute the words "Hello" and "hello" starting a line with the word "Greetings."
`:1,$s/^ *//`	Eliminate one or more spaces at the beginning of all the lines in the file.

Exercise 7.9

Create a file that contains the words "UNIX," "Linux," "Windows," and "DOS." Be sure that some of the lines in this file end with those words. Replace the string `Windows` with `UNIX` in the whole document as you edit it with the vim editor. What command(s) did you use?

Exercise 7.10

As you edit the document in Exercise 7.9, in vim, run the command `:1,$s/DOS\./LINUX\./gp`. What did the command do to the document?

7.6 SEARCHING FILES

UNIX has powerful utilities for file searching that allow you to find lines in text files that contain a particular expression, string, or pattern. For example, if you have a large file that contains the records for a company's employees, one per line, you might want to search the file for line(s) containing information on John Johnsen. The utilities that allow file searching are `grep`, `egrep`, and `fgrep`. The following is a brief description of these utilities.

SYNTAX

```
grep [options] pattern [file-list]
egrep [options] [string] [file-list]
fgrep [options] [expression] [file-list]
```

Purpose: Search the files in **file-list** for the given pattern, string, or expression; if no **file-list**, take input from standard input

Output: Lines containing the given pattern, string, or expression on standard output

Commonly used options/features:

`-c`	Print the number of matching lines only
`-i`	Ignore the case of letters during the matching process
`-l`	Print only the names of files with matching lines
`-n`	Print line numbers along with matched lines
`-s`	Useful for shell scripts; suppresses error messages (the *return status* is set to zero for success and nonzero for no success—see Chapter 10)
`-v`	Print nonmatching lines
`-w`	Search for the given pattern as a string

Of the three, the fgrep command is the fastest but most limited; egrep is the slowest but most flexible, allowing full use of regular expressions; and grep has reasonable speed and is fairly flexible in terms of its support of regular expressions. In the following sessions, we illustrate the use of these commands with some of the options shown in the description. We use the same **students** file in these sessions that we used in describing the sort utility in Section 7.4. We display the file by using the cat command.

```
% cat students
John Johnsen       john.johnsen@tp.com      503.555.1111
Hassaan Sarwar     hsarwar@k12.st.or        503.444.2132
David Kendall      d_kendall@msnbc.org      229.111.2013
John Johnsen       j.johnsen@psu.net        301.999.8888
Ibraheem Sarwar    ibraheem@abc.sci.com     222.123.4567
Kelly Kimberly     kellyk@umich.gov         555.123.9999
Maham Sarwar       smsarwar@k12.st.or       713.888.0000
Jamie Davidson     j.davidson@uet.edu       515.001.1212
Sandy Khan   sandy.khan@isu.edu             515.101.9009
Nabeel Sarwar      n.sarwar@xyz.net         434.555.1212
%
```

The most common and simple use of the grep utility is to display the lines in a file containing a particular string, word, or pattern. In the following session, we display those lines in the **students** file that contain the string Sarwar. The lines are displayed in the order they occur in the file.

```
$ grep Sarwar students
Hassaan Sarwar     hsarwar@k12.st.or        503.444.2132
Ibraheem Sarwar    ibraheem@abc.sci.com     222.123.4567
Maham Sarwar       smsarwar@k12.st.or       713.888.0000
Nabeel Sarwar      n.sarwar@xyz.net         434.555.1212
%
```

The grep command can be used with the -n option to display the output lines with line numbers. In the following session, the lines in the **students** file containing the string John are displayed with line numbers.

```
% grep -n John students
1: John Johnsen    john.johnsen@tp.com      503.555.1111
4: John Johnsen    j.johnsen@psu.net        301.999.8888
%
```

You can use the grep command to search a string in multiple files with regular expressions and shell metacharacters. In the following session, grep searches for the string "include" in all the files in the present working directory that end with **.c** (C source files). Note that the access permissions for **server.c** were set so that grep couldn't read it; the user running the command did not have read permission for the **server.c** file.

```
$ grep -n include *.c
client.c: 21:     #include    <stdio.h>
client.c: 22:     #include    <ctype.h>
client.c: 23:     #include    <string.h>
lab1.c:   13:     #include    <stdio.h>
grep: can't open server.c
$
```

You can also use the `grep` command with the `-l` option to display the names of files in which the pattern occurs. However, it does not display the lines that contain the pattern. In the following session, the **~/States** directory is assumed to contain one file for every US state, and this file is assumed to contain the names of all the cities in the state (e.g., Portland). The `grep` command, therefore, displays the names of files that contain the word "Portland"—that is, the names of states that have a city called Portland.

```
$ grep -l Portland ~/States
Maine
Oregon
$
```

Certain characters are treated specially by both `shell` and `grep`. Therefore, in order to make sure that `shell` passes the desired regular expression to `grep`, you need to enclose the regular expression in single or double quotes. You can pass quote a character by using backslash (`\`), unless the character is newline (n). A single quote (`'`) quotes every character except itself. A double quote (`"`) quotes every character except ", `$`, |, or `'`. Thus, you may replace `"` with `'` in any command that uses regular expressions enclosed in `"` and the command would work, but not vice versa. In the following sessions, we use single and double quotes interchangeably. Expressions enclosed in single and double quotes are also passed verbatim to `grep` and `egrep`.

The following command displays the lines in the **students** file that start with the letters `A` through `H`. In the command, `^` specifies the beginning of a line.

```
% grep '^[A-H]' students
Hassaan Sarwar    hsarwar@k12.st.or       503.444.2132
David Kendall     d_kendall@msnbc.org     229.111.2013
%
```

The following command displays the lines from the **students** file that contain at least eight consecutive lowercase letters.

```
% grep '[a-z]\{8\}' students
Ibraheem Sarwar   ibraheem@abc.sci.com    222.123.4567
Maham Sarwar      smsarwar@k12.st.or      713.888.0000
Jamie Davidson    j.davidson@uet.edu      515.001.1212
%
```

The character sequence \< is used to indicate the start of a word. Single (or double) quotes are used to ensure that the shell does not interpret any letter in the pattern as a shell metacharacter, as in `'\<Ke'` or in `"\<Ke"`. Thus, the following command displays the lines that contain a word starting with the string "Ke."

```
% grep "\<Ke" students
David Kendall     d_kendall@msnbc.org     229.111.2013
Kelly Kimberly    kellyk@umich.gov        555.123.9999
%
```

By using the regular expression `"\<K"`, the output of the `grep` command displays lines that contain words starting with the letter `K`. Thus, the output of the command also includes the line for Sandy Khan, as follows.

```
% grep "\<K" students
David Kendall     d_kendall@msnbc.org     229.111.2013
Kelly Kimberly    kellyk@umich.gov        555.123.9999
Sandy Khan        sandy.khan@isu.com      515.101.9009
%
```

The string \> is the end of the word anchor. Thus, the following command displays the lines that contain words that end with "net." If we replace the string `net` with the string `war`, what would be the output of the command?

```
% grep 'net\>' students
John Johnsen      j.johnsen@psu.net       301.999.8888
Nabeel Sarwar     n.sarwar@xyz.net        434.555.1212
%
```

In the following command, the regular expression **"**`Kimberly|Nabeel`**"** is used to have `egrep` display the lines, and their numbers, that contain either "Kimberly" or "Nabeel." Note that the regular expression uses the pipe symbol `(|)` to logically OR the two strings.

```
% egrep -n "Kimberly\|Nabeel" students
6:Kelly Kimberly  kellyk@umich.gov        555.123.9999
10:Nabeel Sarwar  n.sarwar@xyz.net        434.555.1212
%
```

The `egrep -v Kimberly\|Nabeel students` command would also produce the same result because the pipe character has been escaped using \|.

You can use the `-v` option to display the lines that do not contain the string specified in the command. The following command produces all the lines not containing the words "Kimberly" and "Nabeel."

```
% egrep -v Kimberly\|Nabeel students
John Johnsen      john.johnsen@tp.com     503.555.1111
Hassaan Sarwar    hsarwar@k12.st.or       503.444.2132
David Kendall     d_kendall@msnbc.org     229.111.2013
John Johnsen      j.johnsen@psu.net       301.999.8888
Ibraheem Sarwar   ibraheem@abc.sci.com    222.123.4567
Maham Sarwar      smsarwar@k12.st.or      713.888.0000
Jamie Davidson    j.davidson@uet.edu      515.001.1212
Sandy Khan        sandy.khan@isu.edu      515.101.9009
%
```

The following command displays the lines in the **students** file that start with the letter `J`. Note the use of ^ to indicate the beginning of a line.

```
% egrep "^J" students
John Johnsen      john.johnsen@tp.com     503.555.1111
John Johnsen      j.johnsen@psu.net       301.999.8888
Jamie Davidson    j.davidson@uet.edu      515.001.1212
%
```

The following command displays the lines in the **students** file that start with the letters `J` or `K`. Note that `^J` and `^K` represent lines starting with the letters `J` and `K`.

```
% egrep "^J|^K" students
John Johnsen      john.johnsen@tp.com     503.555.1111
John Johnsen      j.johnsen@psu.net       301.999.8888
Kelly Kimberly    kellyk@umich.gov        555.123.9999
Jamie Davidson    j.davidson@uet.edu      515.001.1212
%
```

The `egrep \^J|\^K students` command could produce the same result. However, the `egrep "^J|^K" students` command uses simpler syntax. As a rule of thumb, if you need to escape multiple special characters in a regular expression, enclose the regular expression in single or double quotes, as the case may be.

In the following in-chapter exercises, you will use the commands of the `grep` family to understand their various characteristics.

Exercise 7.11

Give a command for displaying the lines in the **~/Personal/Phones** file that contain the words starting with the string `David`.

Exercise 7.12

Give a command for displaying the lines in the **~/Personal/Phones** file that contain phone numbers with area code 212. Phone numbers are stored as xxx-xxx-xxxx, where x is a digit from 0 to 9.

Exercise 7.13

Display the names of all the files in your home directory that contain the word “main” (without quotes).

7.7 CUTTING AND PASTING

You can process files that store data in the form of tables in UNIX by using the `cut` and `paste` commands. A table consists of lines, each line comprises a record, and each record has a fixed number of fields. Tabs or spaces usually separate fields, although any field separator can be used. The `cut` command allows you to cut one or more fields of a table in one or more files and send them to standard output. In other words, you can use the `cut` command to slice a table vertically in a file across field boundaries. The following is a brief description of the command.

SYNTAX

```
cut -blist [-n] [file-list]
cut -clist [file-list]
cut -flist [-dchar] [-s] [file-list]
```

Purpose: Cut out fields of a table in a file
Output: Fields cut by the command
Commonly used options/features:

`-b list`	Treat each byte as a column and cut bytes specified in the **list**
`-c list`	Treat each character as a column and cut characters specified in the **list**
`-d char`	Use the character **`char`** instead of the <**`Tab`**> character as field separator
`-f list`	Cut fields specified in the **list**
`-n`	Do not split characters (used with **`-b`** option)
`-s`	Do not output lines that do not have the delimiter character

Here, **list** is a comma-separated list with `-` used to specify a range of bytes, characters, or fields. The following sessions illustrate some of the commonly used options and features of the `cut` command. In this section, we use the file **student_addresses**, whose contents are displayed by the `cat` command.

```
% cat student_addresses
John  Doe   jdoe@xyz.com       312.111.9999      312.999.1111
Pam   Meyer meyer@uop.pk       666.222.1212      666.555.1212
Jim   Davis jamesd@aol.org     713.999.5555      713.413.0000
Jason Kim   j_kim@up.org       434.000.8888      434.555.2211
Amy   Nash  nash@state.gov     888.111.4444      888.827.3333
%
```

The file has five fields numbered 1–5, from left to right: first name, last name, e-mail address, home phone number, and work phone number. Although we could have used any character as the field separator, we chose the <`Tab`> character to give a “columnar” look to the table and the output of the following `cut` and `paste` commands. You can display a table of first and last names by using the `-f` option. Note that `-f1,2` specifies the first and the second fields of the **student_addresses** file.

```
% cut -f1,2 student_addresses
John  Doe
Pam   Meyer
Jim   Davis
Jason Kim
Amy   Nash
%
```

We generate a table of names (first and last) and work phone numbers by slicing the first, second, and fifth fields of the table in the **student_addresses** file.

```
$ cut -f1,2,5 student_addresses
John  Doe   312.999.1111
Pam   Meyer 666.555.1212
Jim   Davis 713.413.0000
Jason Kim   434.555.2211
Amy   Nash  888.827.3333
%
```

To generate a table of names and e-mail addresses, we use the following command. Here, `-f1-3` specifies fields 1–3 of the **student_addresses** file.

```
% cut -f1-3 student_addresses
John  Doe   jdoe@xyz.com
Pam   Meyer meyer@uop.pk
Jim   Davis jamesd@aol.org
Jason Kim   j_kim@up.org
Amy   Nash  nash@state.gov
%
```

We recommend that you run this command on your machine to determine whether the desired output is produced. If the desired output is not produced, you have not used the `<Tab>` character as the field separator for some or all of the records. In such a case, correct the table and try the command again.

In the preceding sessions, we have used the default field separator, the `<Tab>` character. Depending on the format of your file, you can use any character as a field separator. For example, as we discussed in Chapter 4, the **/etc/passwd** file uses the colon character (:) as the field separator. You can therefore use the `cut` command to extract information such as the login name, real name, group ID, and home directory for a user. Because the real name, login name, and home directory are the fifth, first, and sixth fields, respectively, the following command can be used to generate a table of names of all users, along with their login IDs and home directories. The first two lines of the output are for comments, as they start with #.

```
$ cut -d: -f5,1,6 /etc/passwd
# $FreeBSD$
#
root:Charlie &:/root
toor:Bourne-again Superuser:/root
daemon:Owner of many system processes:/root
...
sshd:Secure Shell Daemon:/var/empty
smmsp:Sendmail Submission User:/var/spool/clientmqueue
...
sarwar:Syed Mansoor Sarwar:/home/sarwar
...
$
```

Note that the `-d` option is used to specify `:` as the field separator, and it is also displayed as the field separator in the output of the command. For blank delimited files, use one or more space characters (blanks) after `–d\`, as shown in the following example. The `cat sample` command is used to display the blank delimited file, called **sample**, and the `cut –d\  -f1,6 sample` command is used to display fields 1 and 6 of this file.

```
$ cat sample
1 John CS Senior john@net2net.com 3.45
2 Jane CS Junior jane@net2net.com 3.76
3 Sara CS Senior sara@net3net.vom 3.33
$ cut -d  -f1,6 sample
1 3.45
2 3.76
3 3.33
$
```

The `paste` command complements the `cut` command; it concatenates files horizontally (the `cat` command concatenates files vertically). Hence, this command can be used to paste tables in columns. The following is a brief description of the command.

SYNTAX

`paste [options] file-list`

Purpose: Horizontally concatenate files in **file-list**; use standard input if **-** (i.e., a hyphen) is used as a file

Output: Files in **file-list** pasted (horizontally concatenated)

Commonly used options/features:

`-d list` Use **list** characters as line separators; <**`Tab`**> is the default character

Consider the file **student_records**, which contains student names (first and last), major, and current GPA.

```
% cat student_records
John  Doe   ECE   3.54
Pam   Meyer CS    3.61
Jim   Davis CS    2.71
Jason Kim   ECE   3.97
Amy   Nash  ECE   2.38
%
```

We can combine the two tables, **student_records** and **student_addresses_shortened**, horizontally and generate another by using the `paste` command in the following session. In order to keep the resultant table small, we have used a shortened version of the original **student_addresses** file called **student_addresses_shortened**, which contains the first name, last name, and work phone number only, as shown below. We generated this table by using the `cut -f1,2,5 student_addresses > student_addresses_shortened` command. Note that the output of the `paste` command is displayed on the display screen and is not stored in a file. The resultant table has seven fields.

```
% cat student_addresses_shortened
John  Doe   312.999.1111
Pam   Meyer 666.555.1212
Jim   Davis 713.413.0000
Jason Kim   434.555.2211
Amy   Nash  888.827.3333
% paste student_records student_addresses_shortened
John  Doe   ECE   3.54  John  Doe   312.999.1111
Pam   Meyer CS    3.61  Pam   Meyer 666.555.1212
Jim   Davis CS    2.71  Jim   Davis 713.413.0000
Jason Kim   ECE   3.97  Jason Kim   434.555.2211
Amy   Nash  ECE   2.38  Amy   Nash  888.827.3333
%
```

Suppose that you want to use the **student_addresses_shortened** and **student_records** tables to generate and display a table that has student names, majors, and home phone numbers. You may do so in one of two ways. When you use the first method, you cut the appropriate fields of the two tables, put them in separate files with the fields in the order you want to display them, paste the two tables in the correct order, and remove the tables. The following session illustrates this procedure and its result. Note that the new table is not saved as a file when the following commands are executed. If you want to save the new table in a file, use the `paste table1 table2 > students_table` command. The **students_table** contains the columns of **table1** and **table2** (in that order) pasted together.

```
% cut -f1-3 student_records > table1
% cut -f4 student_addresses > table2
% paste table1 table2
John  Doe   ECE   312.111.9999
Pam   Meyer CS    666.222.1212
Jim   Davis CS    713.999.5555
Jason Kim   ECE   434.000.8888
Amy   Nash  ECE   888.111.4444
% rm table1 table2
%
```

The procedure just outlined is expensive in terms of space and time because you have to execute four commands, generate two temporary files (**table1** and **table2**) on disk, and remove these temporary files after the desired table has been displayed. You can use a different method to accomplish the same thing with the following command.

```
% paste student_records student_addresses | cut -f1-3,8
John  Doe   ECE   312.111.9999
Pam   Meyer CS    666.222.1212
Jim   Davis CS    713.999.5555
Jason Kim   ECE   434.000.8888
Amy   Nash  ECE   888.111.4444
%
```

Here, you first combine the tables in the two files into one table with nine columns by using the `paste student_records student_addresses` command and then displaying the desired table by using the `cut -f1-3,8` command. Clearly, this second method is the preferred way to accomplish the task because no temporary files are created and only one command is needed. If you want to save the resultant table in the **students_table** file, use the command `paste student_records student_addresses | cut -f1-3,8 > students_table`.

7.8 ENCODING AND DECODING

E-mail messages are transported in clear (plain) text, and some e-mail systems are fussy about certain characters contained in the body of the message, such as the tilde character (~) in the first column for the `mail` and `mailx` utilities. This is a serious problem for mail systems, such as `mail`, that do not have convenient support for attachments when you need to attach items such as pictures, videos, or executable programs (binaries). You can use the UNIX-to-UNIX encode (`uuencode`) utility to convert such a file into a format that contains printable ASCII characters only, with a letter in the first column, and then e-mail this file's contents as the body of your e-mail message. The receiver can save the uuencoded contents in the e-mail body into a file and use the `uudecode` utility to convert this file to the original format. In this section, we discuss these two utilities, starting with their brief descriptions.

SYNTAX

```
uuencode [option] [source-file] decode_label
uuencode [-o output-file] [source-file] decode_label
```

Purpose: Convert source file from binary to ASCII
Output: First syntax: the encoded version of **source-file** to standard output
Second syntax: the encoded version of **source-file** to **output-file**
Commonly used options/features:
`-m` Use Base64 method of encoding instead of the standard algorithm
uudecode [options] [encoded-file]
Purpose: Decode **encoded-file** from ASCII to binary
Output: The binary version of **encoded-file** into a file called **decode-label**, the second parameter of the `uuencode` command
Commonly used options/features:
`-i` Do not overwrite files
`-p` Send the decoded (binary) version of the file to standard output
`-o output _ file` Send the decoded (binary) version of the file to **output_file**

The `uuencode` command sends the encoded (ASCII) version of the file to standard output. The command takes input from standard input if no **source-file** is specified in the command. The output has **decode_label** in the header (first line) of the ASCII version.

The `uudecode` utility recreates the original file from the uuencoded (ASCII) file and puts it in a file called **decode_label**. With the `-p` option, the command sends the binary version to standard output. This option uses the `uudecode` command in a pipeline (see Chapter 9). Both the uuencode and uudecode commands retain the original files that they translate. The diagram shown in Figure 7.1 illustrates the process of uuencoding and uudecoding.

You can redirect the output to a file by using the > symbol, as shown in the `uuencode a.out alarm.out > sarwar.out` command in the following session. When you do so, the encoded output goes to the file **sarwar.out** with a **decode_label** of **alarm.out**.

```
% cat a.out
ELF   >'@@?
             @@@@@@@?@@@@ ''( ''?@@00P?@$$Q?t/libexec/ld-elf.
so.FreeBSD>FreeBSD
```

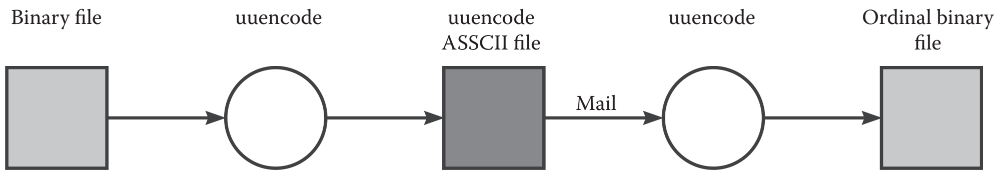


FIGURE 7.1 The process of uuencoding and uudecoding.

```
V?g)]i1N?k??|CE??Pv??qX
????
'n???
'C?
u???
'libc++.so.1_Jv_RegisterClasseslibcxxrt.so.1libm.so.5libc.
so.7__prognameenvironprintf_init_tlsatexit_edata__bss_start_
endFBSD_1.09?(z??
'?
'?
'?
'H??g??H???5? ?%? @?%? h??????%? h??????%~ h??????%v h????UH??AWAV
AUATSPI??M?4$Ic?M?|?H?=c uL?=Z I?E??~4I?$H??t+?fffff.?H??H?
?H?????/t??u?'H??t
H???G?????P????'H??????@?(????'?'H)?I??I???I??=I?I??M??t11?fffff.?
H??'H??r
                                                               D??L??
...
% uuencode -o sarwar.out a.out alarm.out
% head sarwar.out
begin 755 alarm.out
M?T5,1@(!'0D''''''''''('/@'!''''8'5'''''''!''''''''''',@+''''
M'''''''''$''.''('$'''''9''8''''%''''0'''''''''!''$'''''''$''
M0'''''''P'$'''''''#''0''''''''@''''''''''P''''0''''''@''''''
M'''"0''''''''')''''''''5''''''''''!4''''''''''0''''''''''!''''
M!0''''''''''''''''!''''''''''''$''''''',0(''''''''Q'@'''''''''
M'"'''''''''$''''&''''R'@''''''''#("&''''''',@(8'''''''''(''''
M'''H'@''''''''''(''''''''@''''8''''#P"''''''''/'(8''''''''A@
M'''''''"@'0'''''''*'!'''''''''"''''''''''$''''!''''!@"''''''''
M&')''''''''8'D'''''''#''''''''''',''''''''''$'''''''''%#E=&0$
% uudecode sarwar.out
% cat alarm.out
ELF   >'@@?
             @@@@@@@?@@@@ ''( ''?@@00P?@$$Q?t/libexec/ld-elf.
so.FreeBSD>FreeBSD

V?g)]i1N?k??|CE??Pv??qX
????
'n???
'C?
u???
'libc++.so.1_Jv_RegisterClasseslibcxxrt.so.1libm.so.5libc.
so.7__prognameenvironprintf_init_tlsatexit_edata__bss_start_
endFBSD_1.09?(z??
'?
'?
'?
```

```
'H??g??H???5? ?%? @?%? h??????%? h??????%~ h??????%v h????UH??AWAV
AUATSPI??M?4$Ic?M?|?H?=c uL?=Z I?E??~4I?$H??t+?fffff.?H??H?
?H?????/t??u?'H??t
H???G?????P????'H??????@?(????'?'H)?I??I???I??=I?I??M??t11?fffff.?
H??'H??r
                                                          D??L??
...
% ls -l a.out sarwar.out
-rwxr-xr-x  1 sarwar  faculty  7121 Jul 13 12:25 a.out
-rw-r--r--  1 sarwar  faculty  9840 Aug 31 11:17 sarwar.out
%
```

Note the label **alarm.out** on the first line of the uuencoded file **sarwar.out**. The `uudecode` command translates the file **sarwar.out** and puts the original in the file **alarm.out**. As expected, **a.out** and **alarm.out** contain the same data. If you want to recreate the original file in **a.out**, use **a.out** as the label in the `uuencode` command, as in `uuencode -o sarwar.out a.out a.out`. The `uudecode sarwar.out` command produces the original binary file in the **a.out** file.

The output generated by `uuencode` is about 38% *larger* than the original file. Thus, for efficient use of the network bandwidth, you should compress binary files before uuencoding them and uncompress them after uudecoding them. Doing so is particularly important for large picture files or files containing multimedia data, such as videos.

7.9 FILE ENCRYPTION AND DECRYPTION

We briefly described *encryption* and *decryption* of files in Chapter 5. Here, we describe these processes in more detail with the help of the UNIX command `crypt`.

Recall that encryption is a process by which a file is converted to a form completely different from its original version and that the transformed file is called an *encrypted* file; the reverse process of transforming the encrypted file to its original form is known as decryption. Figures 7.2 and 7.3 illustrate these processes.

You encrypt files to prevent others from reading them. You can also encrypt your e-mail messages to prevent hackers from understanding your message even if they are able to tap a network as your message travels through it. On a UNIX system, you can use the `crypt` command to encrypt and decrypt your files. The following is a brief description of the command.

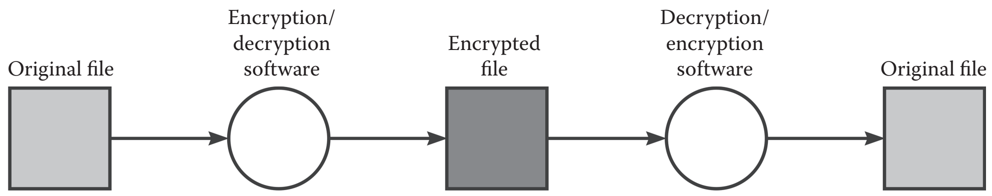

Original file
Encryption/ decryption software
Encrypted file
Decryption/ encryption software
Original file

FIGURE 7.2 The process of encryption and decryption.

SYNTAX

`crypt [options]`

Purpose: Encrypt (decrypt) standard input and send it to standard output
Output: Encrypted (decrypted) version of the input text
Commonly used options/features:

`key`	Password to be used to perform encryption (and decryption)
`-k`	Use the value of the environment variable CRYPTKEY

By default, the `crypt` command takes input from standard input and sends its output to standard output. The optional argument *key* is a password used in the encryption and decryption processes. The command is used mostly with actual files, not keyboard input, so the commonly used syntax for the `crypt` command is:

```
crypt key < original_file > encrypted_file
```

To decrypt an encrypted file, the process is reversed according to the syntax:

```
crypt key < encrypted_file > original_file
```

Remember that the key must be the same for both commands. The user chooses the file name after the greater-than symbol (>). Multiple files can be specified as source files, but the command encrypts only the first file and ignores the remaining files. The semantics of these commands are shown in Figure 7.3. Note that the original file (the file to be encrypted) remains intact and must be explicitly removed from the system after the encrypted version has been generated.

The following session illustrates the use of `crypt` for encrypting and decrypting a file called **memo1**. The `crypt !hskr45#$ < memo1 > secret_memo1` command encrypts **memo1** (in the present working directory) by using `hskr45#$` as the key and puts the encrypted version in the file **secret_memo1**, also in the present working directory. The `cat` commands before and after the `crypt` command show the contents of the original and encrypted files. Note that the contents of the encrypted file, **secret_memo1**, are not readable, which is the objective. Also note that your shell prompt might get messy after the `cat secret_memo1` command has completed its execution. If this happens, you should logout and login again.

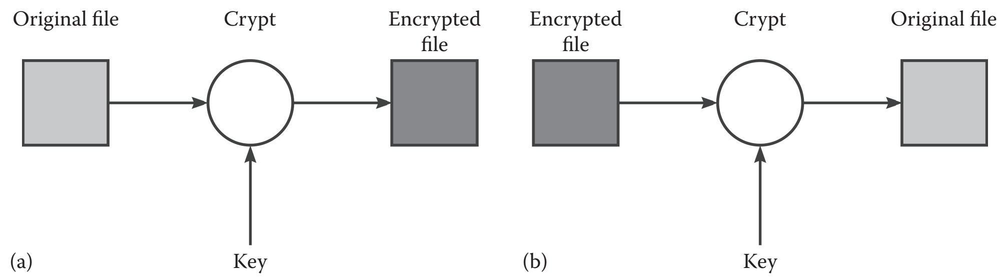


FIGURE 7.3 Encryption and decryption of a file by using the `crypt` command.

```
$ cat memo1
Dear Jim:
This is to inform you that the second quarter earnings do not
look good. This information will be made public on Monday next
week when The Wall Street Journal reports the company earnings.
Of course, the company stock will take a hit, but we need to keep
the morale of our employees high. I am calling a meeting of the
vice presidents tomorrow morning at 8:00 to talk about this
issue in the main conference room. See you then. Make sure this
information does not get out before time.
Nadeem
$ crypt hskr45#$ < memo1 > secret_memo1
$ cat secret_memo1
.??&??
      ?k7??6??I??<~?eb?'!ج'I?w??
]?b?e□???_??R?y?????7xI<???Y-A4?5x?1?#?98
      2GF≠?WF?????$┴□?V????*┐????┼°␍?W??S%?┼??????????
┼C?A??±?"??·□????␍?␉???□?┼??KOBM?Q┬D┤□?WF?M␊*W[O?K???8?┤
␤&7????─

□T?≤W?≠*??\T□0␌W['T??├?E???␉??$??ﷺ┤
??\?▓?8M°^????≤??)?\??٦W?\??□K□? ≤?????2WA???#???┬???S????N?°?

??8?????□?£≠?4┤?°H??┐ú?.□?┘?X?≤□@???[TY??␉S?Q?□
                          B?C????N?&????_?□*?├□??□?┐?_ (?<?□"?@??

...
%
```

The command for decrypting the file **secret_memo1** and putting the original version in the file **original_memo1** is as follows. The cat command confirms that the original file has been restored.

```
$ crypt hskr45#$ < secret_memo1 > original_memo1
$ cat original_memo1
Dear Jim:
This is to inform you that the second quarter earnings do not
look good. This information will be made public on Monday next
week when The Wall Street Journal reports the company earnings.
Of course, the company stock will take a hit, but we need to keep
the morale of our employees high. I am calling a meeting of the
vice presidents tomorrow morning at 8:00 to talk about this
issue in the main conference room. See you then. Make sure this
information does not get out before time.
Nadeem
$
```

Again, the `crypt` command does not remove the file that it encrypts (or decrypts), and it is your responsibility to remove the original file after encrypting it. Note also that the encryption algorithm used by the `crypt` command is not the same as that used for encrypting user passwords, as found in the **/etc/passwd** or **/etc/shadow** files. Also, even a superuser cannot decrypt an encrypted file without having the correct key.

The `crypt` command uses an encryption technique that was used in the German Enigma machine during World War II, although some of the parameters used in `crypt` make the output generated by the command more difficult to decipher than that of the Enigma machine. However, the methods of attack are well known for such machines. The level of security provided by `crypt` is therefore minimal, and the command must not be used on documents that require an extremely high level of security. However, an average user is not familiar with these attack methods, and the use of `crypt` is a fairly decent method of protecting your files.

If your work requires documentation and communication at higher levels of security, you should use the `openssl` tool. It supports the `crypto` library, which allows you to use several cryptographic functions, including the following:

- Encryption and decryption with ciphers
- Creation and management of public and private keys
- Public-key cryptographic operations
- Handling of encrypted e-mail

The coverage of `openssl` and `crypto` is beyond the scope of this book. You can learn more about them by using the man `openssl` and man `crypto` commands.

The following in-chapter exercises will give you practice using the `crypt`, `cut`, `paste`, `uudecode`, and `uuencode` commands and help you to understand their semantics with a hands-on session.

Exercise 7.14

Create the **student_addresses** and **student_records** files used in Section 7.7. Then run the `cut` and `paste` commands described in this section to see how these commands work.

Exercise 7.15

Copy the executable code for a command from the **/usr/bin** directory and `uuencode` it. Run the `ls -l` command and report the size of the encoded file. Then `uudecode` the encoded file to convert it back to the original file.

Exercise 7.16

Try the sessions for the `crypt` command given in this section on your system.

SUMMARY

Several advanced operations have to be performed on text and nontext files from time to time. These operations include compressing and uncompressing file contents, sorting files, searching for files and commands in the file system structure, searching files for certain strings or patterns, performing database-like operations of cutting fields from a table or pasting tables together, transforming non-ASCII files to ASCII, and encrypting and decrypting files. Several tools are available in UNIX for performing these tasks.

Some of these tools have the ability to specify a set of strings by using a single character string comprising of constants and operators, called regular expressions. Commonly known as search patterns, regular expressions are used to search file contents for desired strings. The utilities that allow the use of regular expressions are `awk`, `ed`, `egrep`, `grep`, `sed`, and `vim`. In this chapter, we described regular expressions and their use in `vim`, `egrep`, and `grep`.

The `compress` and `gzip` commands can be used to compress and uncompress files, with `gzip` being the more flexible of the two. The `uncompress`, `gunzip`, and `gzip -d` commands can be used to uncompress files compressed with the `compress` and `gzip` commands, respectively. The `gzexe` command can be used to compress executable files and the `gzexe -d` command can be used to uncompress them. Files compressed with `gzexe` can be executed without explicitly uncompressing them. The `zcat` and `zmore` commands can be used to display compressed files without explicitly uncompressing them.

The `sort` command can be used to sort text files. Each line comprises a record with several fields, and the number of fields in all the lines in the file is the same. Text files can also be processed like tables by using the `cut` and `paste` commands that allow cutting of columns in a table and pasting of tables, respectively. The `sort`, `cut`, and `paste` commands can be combined via a pipeline (see Chapter 9) to generate tables based on different sets of criteria.

The `find` and `whereis` commands can be used to search the UNIX file system structure to determine the locations (absolute pathnames) of files and commands. The `find` command, in particular, is very powerful and lets you search for files based on several criteria, such as file size. The `which` command can be used to determine which version of a command executes, in case there are several versions available on a system.

UNIX also provides a family of powerful utilities for searching for strings, expressions, and patterns in text files. These utilities are `grep`, `egrep`, and `fgrep`. Of the three, `fgrep` is the fastest but most limited; `egrep` is the most flexible but slowest of the three; and `grep` is the middle-of-the-road utility—reasonably fast and fairly flexible.

The `uuencode` and `uudecode` utilities are useful in situations when users want to e-mail non-ASCII files such as multimedia files, but the mailing system does not allow attachments. The `uuencode` utility can be used to transform a non-ASCII file into an ASCII file, and `uudecode` can transform the ASCII file back into the original non-ASCII version. Thus, the sender uses the `uuencode` command before e-mailing a non-ASCII file as part of the e-mail body, and the receiver of a uuencoded file uses `uudecode` to convert it back to its original form.

In the UNIX system, the `crypt` command can be used to encrypt and decrypt files that the user wants to keep secret. The techniques for converting an encrypted file back to original are well known. However, the average user is not familiar with these techniques, so the use of `crypt` results in a fairly good scheme for protecting files. If your work requires higher-level secure documentation and communication, you should use the `openssl` and `crypto` tools.

QUESTIONS AND PROBLEMS

1. List five file-processing operations that you consider advanced.
2. What are regular expressions? What do the following regular expressions specify?
   a. `a|bc*`
   b. `(a|b)*`
   c. `a|(b*)c+`
3. Give the `vi` command for replacing all occurrences of the string `DOS` with the string `UNIX` in the whole document that is currently being edited. What are the commands for replacing all occurrences of the strings `DOS` and `Windows` with the string `UNIX` from the lines that start or end with these strings in the document being edited?
4. Give the `vi` command for deleting all four-letter words starting with `B`, `F`, `b`, and `f` in the file being edited.
5. Give the `vi` command for renaming all C source files in a document to C++ source code files. Note: C source files end with **.c** and C++ source files end with **.cpp**.
6. What is file compression? What do the terms *compressed files* and *decompressed files* mean? What commands are available for performing compression and decompression in UNIX? Which are the preferred commands? Why?
7. Take three large files in your directory structure—a text file, a PostScript file, and a picture file—and compress them by using the `compress` command. Which file was compressed the most? What was the percentage reduction in file size? Compress the same files by using the `gzip` command. Which resulted in better compression, `compress` or `gzip`? Uncompress the files by using `uncompress` and `gunzip` commands. Show your work.
8. What is sorting? Give an example to illustrate your answer. Name four applications of sorting. Name the UNIX utility that can be used to perform sorting.
9. Go to http://cnn.com/weather and record the high and low temperatures for the following majors cities in Asia: Kuala Lumpur, Karachi, Tokyo, Lahore, Manila, New Delhi, and Jakarta. In a file called **asiapac.temps**, construct an ASCII table comprising one line per city in the order: city name, high temperature, and low temperature.

The following is a sample line.

```
Tokyo 78 72
```

Give commands to perform the following operations.

a. Sort the table by city name.

b. Sort the table by high temperature.

c. Sort the table by using the city name as the primary and low temperature as the secondary key.

10. For the **students** file in Section 7.6, give a command to sort the lines in the file by using last name only as the sort key.

11. What commands are available for file searching? State the purpose of each.

12. Give the command that searches your home directory and displays pathnames of all the files created after the file **/etc/passwd**.

13. Give a command that searches your home directory and removes all PostScript and .gif files. The command must take your permission (prompt you) before removing a file.

14. On your UNIX system, how long does it take to find all the files that are larger than 1000 bytes in size? What command(s) did you use?

15. What does the command `grep -n '^' student_addresses` do? Assume that **student_addresses** is the same file we used in Section 7.6.

16. Give the command that displays lines in **student_addresses** that start with the letter `K` or have the letter `J` in them. The output of the command should also display line numbers.

17. What do the following commands do?

    a. `grep [A-H] students`

    b. `grep [A,H] students`

18. What would be the alternative of the `grep ' \<Ke' students` command that uses only backslash (`\`) characters only (i.e., no quotes)?

19. What would be the alternative of the `grep ' ^[A-H]' students` command that uses backslashes (`\`) instead of single (or double) quotes?

20. What are the equivalent `grep` commands for the following commands: `egrep`, `fgrep`, `zgrep`, `zegrep`, and `zfgrep`?

21. What reaches the grep command in the following cases?

    a. `grep '<K' students`

b. `grep '\<K' students`

c. `grep <K students`

22. Give a command that displays names of all the files in your **~/courses/ece446** directory that contain the word "UNIX."

23. Give a command that generates a table of user names for all users on your system, along with their personal information. Extract this information from the **/etc/passwd** file.

24. Use the tables **student_addresses** and **student_records** to generate a table in which each row contains last name, work phone number, and GPA.

25. Imagine that you have a picture file **campus.bmp** that you would like to e-mail to a friend. Give the sequence of commands that are needed to convert the file to ASCII form, reduce its size, and encrypt it before e-mailing it.

26. What is the purpose of file encryption? Name the UNIX command that you can use to encrypt and decrypt files. Give the command for encrypting a file called **~/personal/memo7** and store it in a **~/personal/memo_007** file. Be sure that, when the encrypted file is decrypted, it is put back in the **~/personal/memo7** file. Give the command to decrypt the encrypted file.

27. What is the difference between the following:

    a. encryption and encoding?

    b. encoding and compression?

    c. `compress` and `zip`?

28. Store the man pages for `bash`, `sh`, and `csh` in a file called **shell_man_pages**. Use the `gzip` command to compress with different levels of compression. Show your work and identify the command that does the maximum compression, along with the compression ratio.

29. Give an alternative syntax for the following command.

    ```
    uuencode -o sarwar.out a.out alarm.out
    ```

    Hint: Use output redirection, as discussed in Chapter 6.

30. Decode the uuencoded **sarwar.out** file generated by the command given in Problem 25 such that the uudecoded content is saved in **a.out** and not in **alarm.out**.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

CHAPTER 8

File Sharing

Objectives

- To explain different ways of sharing files
- To discuss the UNIX schemes and commands for implementing file sharing
- To describe UNIX hard and soft (symbolic) links in detail and discuss their advantages and disadvantages
- To cover the commands and primitives

  `*`, `~`, `ln`, `ln -f`, `ln -s`, `ls -i`, `ls –l`

8.1 INTRODUCTION

When a group of people work together on a project, they need to share information. If the information to be shared is on a computer system, group members have to share files and directories. For example, authors collaborating on a book or software engineers working on a software project need to share files and directories related to their project. In this chapter, we discuss several ways of implementing file sharing in a computer system. The discussion of file sharing in this chapter focuses on how a file can be accessed from various directories by various users in a UNIX system. Under the topic of *version control* in Chapter 17, we address how members of a team can work on one or more files simultaneously without losing their work.

Several methods can be used to allow a group of users to share files and directories. In this chapter, we describe duplicate shared files, common logins for members of a team, setting appropriate access permissions on shared files, common groups for members in a team, and sharing via links. All these methods can be used to allow a team of users to share files and directories in a UNIX system. Although we describe each of these techniques, the chapter is dedicated primarily to a discussion of sharing via links in a UNIX-based computer system.

8.2 DUPLICATE SHARED FILES

The simplest approach to files is to make copies of these files and give them to all team members. The members can put these copies anywhere in their own accounts (directory structures) and manipulate them in any way they desire. This scheme works well if members of the team are to work on the shared file(s) sequentially, but it has obvious problems if team members are to work on these files simultaneously. In the former case, team members work on one copy of the shared files one by one and complete the task at hand. In the latter case, because the members modify their own copies, the copies become inconsistent and no single copy of the shared files reflects the work done by all the team members. This outcome defeats the purpose of sharing.

8.3 COMMON LOGINS FOR TEAM MEMBERS

In this scheme, the system administrator creates a new user group comprising the members of a team and creates for them a new account to which they all have access; that is, they all know the login name and password for the account. The team owns all the files and directories created by any team member under this account and everyone has access to them.

It is a simple scheme that works quite well, particularly in situations in which the number of teams is small and teams are stable; that is, they stay together for long periods of time. Such is the case for teams of authors writing a book or programming teams working on large software projects that take several months to finish. However, this scheme also has a couple of drawbacks. First, the team's members have to use a separate account for their current project and cannot use their regular accounts to access shared files and directories. Second, the system administrator has to create a new account for every new team formed in the organization. Having to do so could create a considerable amount of extra work for the administrator if the duration of projects is short and new teams are formed for every new project. The scheme could be a real headache for the system administrator in a college-like environment where student teams are formed to work on class projects, resulting in a large number of teams every semester or quarter.

8.4 SETTING APPROPRIATE ACCESS PERMISSIONS ON SHARED FILES

In this scheme, the team members decide to put all shared files under one member's account, and the access permissions on these files are set so that all team members can access them. This scheme works well if *only* this team's members form the user group (recall the discussion of owner, group, and others in Chapter 5) because, if the group has other users in it, they will also have access to the shared files. For example, suppose that two university professors, Art Pohm and Jim Davis, belong to the user group **faculty**. They decide to put their shared files in Davis's account but set the group access permissions to read, write, and execute for all shared files. All the professors in the user group **faculty** then will have the same access permissions to these files, which will pose security problems. In particular, if the information to be shared is a small portion of the total amount of information residing in a member's account (say, two ordinary files out of tens of files and directories that the member owns), the risk of opening the door to all users in a group is too high, and a better technique must be used.

8.5 COMMON GROUPS FOR TEAM MEMBERS

This scheme works just like the preceding one, except that the system administrator creates a new user group consisting of the members of the team only. All team members get individual logins and set access permissions for their files so that they are accessible to other members of the team. This file-sharing scheme is effective and is used often, particularly in conjunction with a version control mechanism (see Chapter 17).

In Section 5.6.3, we show how you can set the sticky bit for a directory to allow a group of users to share files and directories in it, but to ensure that an unprivileged user cannot remove or rename files of other users in that directory.

8.6 FILE SHARING VIA LINKS

As described in Chapter 4, the attributes of a UNIX file are stored in its inode on disk. When a file is opened, its inode is copied into the main memory, allowing speedy access to its contents. In this section, we describe how the use of an inode results in a mechanism that allows you to access a file by using multiple pathnames. System administrators commonly use this scheme to allow access to some files and directories through various other directories. Thus, for example, the home directories of all the users on the system may be accessed through **/home** or **/usr/home**. As discussed in Chapter 4, on PC-BSD (FreeBSD), the commands, tools, and utilities for normal users are located in the **/bin**, **/usr/bin**, and **/usr/local/bin** directories, and commands, utilities, daemons (see Chapter 10), and tools for systems administration are located in the **/sbin**, **/usr/sbin**, and **/usr/local/sbin** directories.

A link is a way to establish a connection between the file to be shared and the directory entries of the users who want to have access to this file. Thus, when we say that a file has *N* links, we mean that the file has *N* directory entries somewhere in the file system hierarchy. The links therefore aid file sharing by providing different access paths to files to be shared. However, the appropriate setting of access permissions on these files controls the level of sharing. You can create links to files to which you do not have any access, but that gets you nowhere. Hence, file sharing via links is accomplished first by creating access paths to shared files by establishing links to them and then by setting appropriate access permissions on these files.

UNIX supports two types of links: *hard links* and *soft/symbolic links*. The `ln` command may be used to create both types of links. The remainder of this chapter discusses methods of creating both types of links and their internal implementation in the UNIX system.

8.6.1 Hard Links

A hard link is a *pointer* to the inode of a file. When a file is created in UNIX, the system allocates a unique inode to the file and creates an entry in the directory in which the file is created. As we discussed in Chapter 4, the directory entry comprises an ordered pair (**inode number**, **filename**). The inode number for a file is used to access its attributes, including its contents on the disk for reading or writing (changing) them (see Chapter 4).

Suppose that you create a file **Chapter3** in your present working directory and the system allocates inode number **53472** to this file; the directory entry for this file would be **(52473, Chapter3)**.

If we assume that your present working directory previously contained files **Chapter1** and **Chapter2**, its logical structure is shown in Figure 8.1 (a). The new file has been highlighted with a gray shade. Figure 8.1 (b) shows the contents of the disk block that contains the present working directory. The connection between this directory entry and the file's contents is shown in Figure 8.1 (c). The inode number in **Chapter3's** directory entry is used

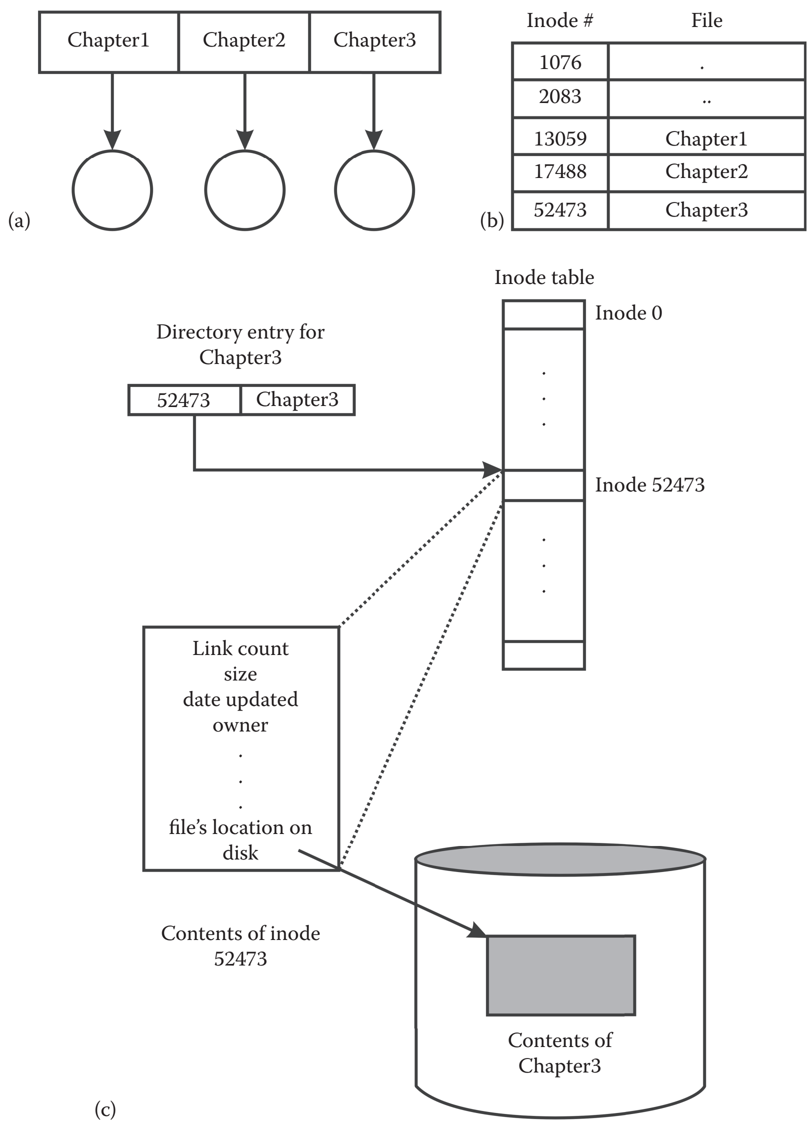


FIGURE 8.1 (a) Logical structure of the current directory; (b) contents of the current directory; (c) relationship between directory entry, inode, and file contents.

to index the inode table in the main memory in order to access the file's inode. The inode contains the attributes of **Chapter3**, including its location on disk.

Here is a brief description of the `ln` command:

SYNTAX

```
ln [options] existing-file new-file
ln [options] existing-file-list directory
```

Purpose: First syntax: Create a link to **existing-file** and name it **new-file**
Second syntax: Create links to the ordinary files in **existing-file-list** in **directory**; links have the same names as the original file

Commonly used options/features:

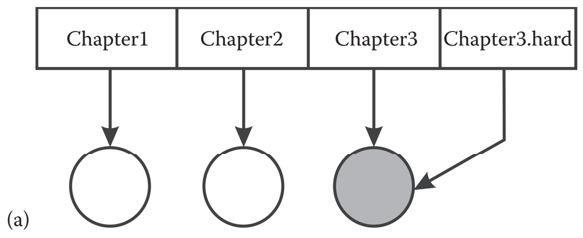
- `-f` Force creation of link; don't prompt if **new-file** already exists
- `-n` Don't create the link if **new-file** already exists
- `-s` Create a symbolic link to **existing-file** and name it **new-file**

The `ln` command without any option creates a hard link to a file, provided the user has execute permission for all the directories in the path leading to the file. The following session illustrates how the `ln` command can be used to create a hard link in the same directory that contains **existing-file**. The only purpose of this example is to illustrate how the `ln` command is used; it isn't representative of how you would establish and use hard links in practice.

```
$ ls -il
13059 -rwx------ 1 sarwar faculty  398 Mar 11 14:20 Chapter1
17488 -rwx------ 1 sarwar faculty 5983 Apr 17 11:57 Chapter2
52473 -rwx------ 1 sarwar faculty 9352 Aug 20 23:09 Chapter3
$ ln Chapter3 Chapter3.hard
$ ls -il
13059 -rwx------ 1 sarwar faculty  398 Mar 11 14:20 Chapter1
17488 -rwx------ 1 sarwar faculty 5983 Apr 17 11:57 Chapter2
52473 -rwx------ 2 sarwar faculty 9352 Aug 20 23:09 Chapter3
52473 -rwx------ 2 sarwar faculty 9352 Aug 20 23:09 Chapter3.hard
$
```

The `ls -il` command shows some of the attributes of all the files in the present working directory, including their inode numbers. The command `ln Chapter3 Chapter3.hard` creates a hard link to the file **Chapter3**; the name of the hard link is **Chapter3.hard**. The system creates a new directory entry (**52473**, **Chapter3.hard**) for **Chapter3** in the present working directory. Thus, you can refer to **Chapter3** by accessing **Chapter3.hard** as well, because both names point to the same file on disk. The second `ls -il` command is used to confirm that **Chapter3.hard** and **Chapter3** are two names for the same file, as both have the same inode number, **52473**, and hence the same attributes. Therefore, when a hard link is created to **Chapter3**, a new pointer to its inode is established in the directory where the link (**Chapter3.hard**, in this case) resides, as illustrated in Figure 8.2.

Note that the output of the `ls -il` command also shows that both **Chapter3** and **Chapter3.hard** have link counts of 2 each. Thus, when a hard link is created to a file, the link count for the file increments by 1. That is, the same file exists in the file structure with two names (i.e., two pathnames). When you remove a file that has multiple hard links, the UNIX system decrements by 1 the link count in the file's inode. If the resultant link count is 0, the system removes



(b)

Inode #	File
1076	.
2083	..
13059	Chapter1
17488	Chapter2
52473	Chapter3
52473	Chapter3.hard

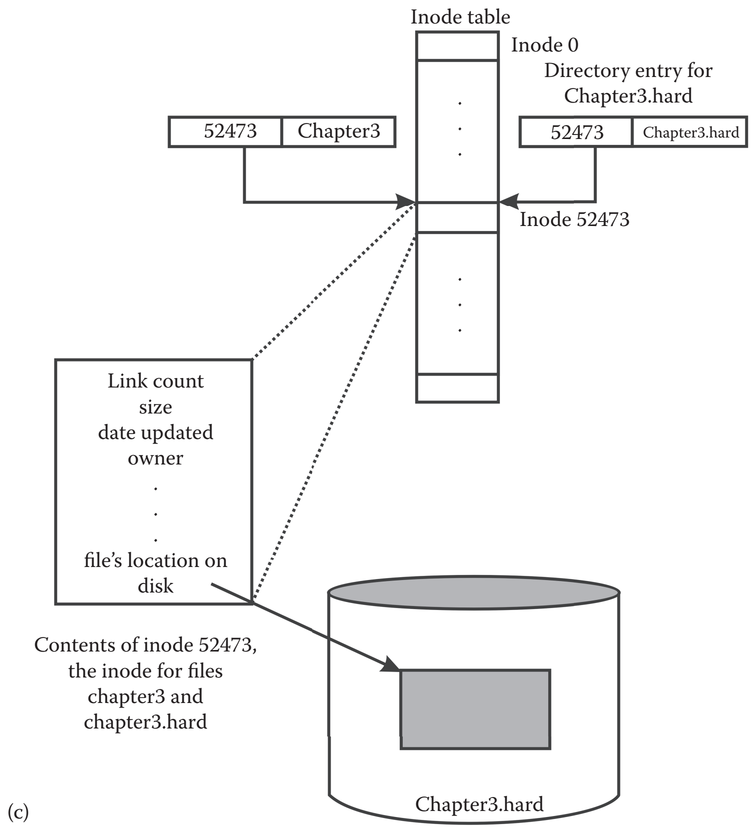


FIGURE 8.2 Establishing a hard link: (a) logical structure of the current directory; (b) contents of the current directory; (c) hard-link implementation by establishing a pointer to the inode of the file.

the directory entry for the file, releases the file's inode and all other kernel resources allocated to the file so they can be reused, and deallocates disk blocks allocated to the file so that they can be used to store other files and/or directories created in the future. If the new link count is not 0, only the directory entry for the removed file is deleted; the file contents and other directory entries for the file (hard links) remain intact. The following session illustrates this point.

```
$ rm Chapter3
$ ls -il
13059 -rwx------ 1 sarwar faculty  398 Mar 11 14:20 Chapter1
17488 -rwx------ 1 sarwar faculty 5983 Apr 17 11:57 Chapter2
52473 -rwx------ 1 sarwar faculty 9352 Aug 20 23:09 Chapter3.hard
$
```

This session clearly shows that removing **Chapter3** results in the removal of the directory entry for this file but that the file still exists on disk and is accessible via **Chapter3.hard**. This link has the inode number and file attributes that **Chapter3** had, except that the link count, as expected, has been decremented from 2 to 1.

The following `ln` command can be used to create a hard link called **memo6.hard** in the present working directory to a file **~/memos/memo6**. The `ls -il` command is used to view attributes of the file before the hard link to it is created.

```
$ ls -il ~/memos/memo6
83476 -rwx------ 1  sarwar  faculty  1673  May 29  11:22  /home/
sarwar/memos/memo6
$ ln ~/memos/memo6 memo6.hard
$
```

After executing the `ln` command, you can run the `ls -il` command to confirm that both files (**~/memos/memo6** and **memo6.hard**) have the same inode number and attributes, as shown in the following session.

```
$ ls -il ~/memos/memo6
83476 -rwx------ 2  sarwar  faculty  1673  May 29  11:22  /home/
sarwar/memos/memo6
$ ls -il memo6.hard
83476 -rwx------ 2 sarwar  faculty  1673  May 29  11:22 memo6.
hard
$
```

The output shows two important things: first, the link count is up by 1; second, both files are represented by the same inode, **83476**. Figure 8.3 shows the hard link pictorially.

In the following session, the `ln` command creates hard links to all nondirectory files in the directory called **~/unixbook/examples/dir1**. The hard links reside in the directory **~/unixbook/examples/dir2** and have the names of the original files in the **dir1** directory. The second argument, **dir2**, must be an existing directory, and you must have execute and

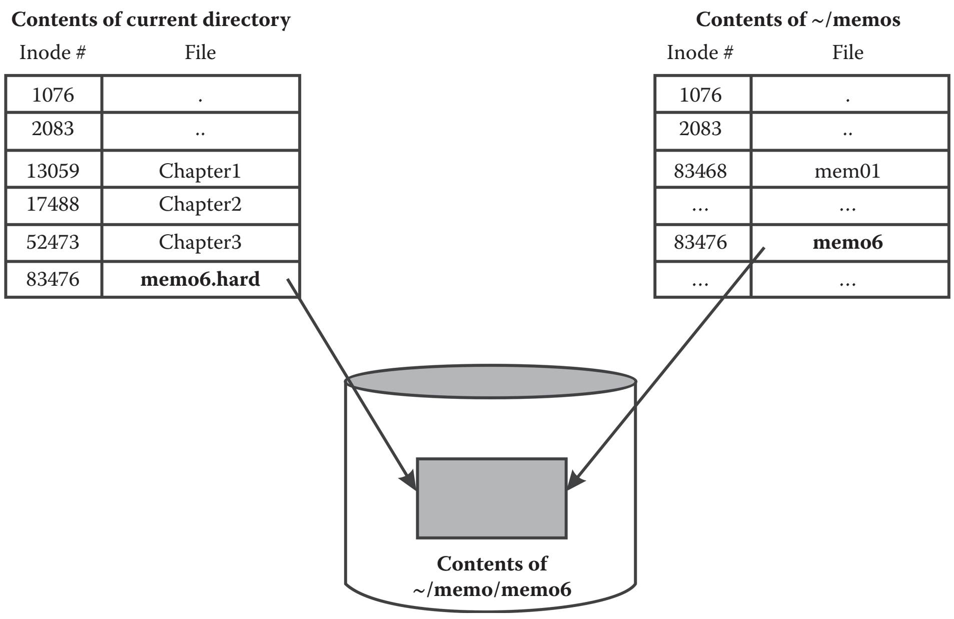


FIGURE 8.3 Pictorial representation of the hard link between **~/memos/memo6** and **memo6.hard** in the current directory.

write permissions to it. Note that the link counts for all the files in **dir1** and **dir2** are 2. The `-f` option is used to force creation of a hard link in case any of the files **f1**, **f2**, or **f3** already exist in the **~/unixbook/examples/dir2** directory.

```
$ cd unixbook/examples
$ more dir1/f1
Hello, World!
This is a test file.
$ ls -l dir1
-rw------- 1 sarwar faculty 35 Jun 22 22:21 f1
-rw------- 1 sarwar faculty 68 May 16 21:03 f2
-rw------- 1 sarwar faculty 94 Jul 11 11:39 f3
$ ln -f ~/unixbook/examples/dir1/* ~/unixbook/examples/dir2
$ ls -l dir1
-rw------- 2 sarwar faculty 35 Jun 22 22:21 f1
-rw------- 2 sarwar faculty 68 May 16 21:03 f2
-rw------- 2 sarwar faculty 94 Jul 11 11:39 f3
$ ls -l dir2
-rw------- 2 sarwar faculty 35 Jun 22 22:21 f1
-rw------- 2 sarwar faculty 68 May 16 21:03 f2
-rw------- 2 sarwar faculty 94 Jul 11 11:39 f3
$ more dir2/f1
Hello, World!
This is a test file.
$
```

You can run the following command to create a hard link in your home directory to the file **/home/sarwar/unixbook/examples/demo1**. The hard link appears as a file **demo1** in your home directory. If **demo1** already exists in your home directory, you can overwrite it with the `-f` option. If **demo1** exists in the home directory and you don't use the `-f` option, an error message is displayed on the screen informing you that the **demo1** file exists. You must have the execute permission for the directories in the pathname **/home/sarwar/unixbook/examples/demo1**, and **demo1** must be a file.

```
$ ln -f /home/sarwar/unixbook/examples/demo1 ~
$
```

The user **sarwar** can run the following command to create a hard link **demo1** in a directory **dir1** in **bob**'s home directory that points to the file **/home/sarwar/unixbook/examples/demo1**. The name of the link in **bob**'s directory is **demo1**, the same as the original file. Figure 8.4 shows the establishment of the link.

```
$ ln -f /home/sarwar/unixbook/examples/demo1 /home/bob/dir1
$
```

The user **sarwar** must have execute permission for **bob**'s home directory and execute and write permissions for **dir1** (the directory in which the link is created). The user **bob** must have proper access permissions for **demo1** in **sarwar**'s directory structure to access this file. Thus, if **sarwar** and **bob** are in the same user group and **bob** needs to edit **demo1**, **sarwar** must set the group access privileges for the file to read and write. Then, **bob** is able to edit **demo1** by using, for example, the `vim demo1` command from his home directory.

The following command accomplishes the same task. Remember that **sarwar** runs this command.

```
$ ln -f ~/unixbook/examples/demo1 /home/bob/dir1
$
```

You can run the following command to create hard links to all nondirectory files in your **~/unixbook/examples** directory. The hard links reside in the **unixbook/examples**

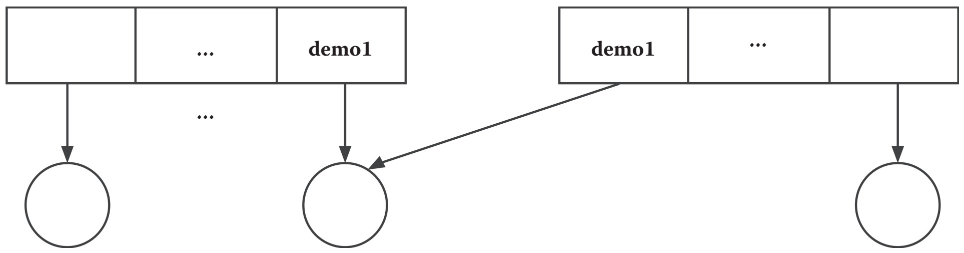


FIGURE 8.4 A hard link between **/home/sarwar/unixbook/examples** and **/home/bob/dir1**.

directory in user **john**'s home directory and have the names of the original files. The user **john** must first create the **unixbook** directory in his home directory and the **examples** directory in his **unixbook** directory. You must have the execute permission for **john's unixbook** directory and execute and write permissions for his **examples** directory for the command to run successfully and accomplish the task.

```
$ ln -f ~/unixbook/examples/* /home/john/unixbook/examples
$
```

8.6.2 Drawbacks of Hard Links

Hard links are the traditional way of *gluing* the file system structure in UNIX, which usually comprises several file systems. Hard links, however, have some problems and limitations that make them less attractive to the average user.

The first problem is that hard links cannot be established between files that are on different file systems. This inability is not an issue if you are establishing links between files in your own directory structure, with your home directory as the top-level directory, or with files in another user's directory structure that is on the same file system as yours. However, if you want to create a hard link between a file (command) in the **/bin** directory and a file in your file structure, it most likely will not work because on almost all systems the **/bin** directory and your directory structure reside on different file systems. The following command illustrates this point. In this example, we try to give the name `del` to the UNIX command `rm` that resides in the directory **/bin**. Because the `rm` command is in one file system (/ on our system) and our directory structure is in another **(/usr/home** on our system), UNIX doesn't allow us to create a hard link, **del**, between a file in the current directory and **/bin/rm**.

```
$ ln /bin/rm del
ln: del: Cross-device link
$
```

This problem also shows up when a file with multiple links is moved to another file system. The following session illustrates this point. The `ls -il` command shows that **Chapter3** and **Chapter3.hard** are hard links to the same file (note the same inode number). The `mv` command is used to move the file **Chapter3** to the **/tmp** directory, which is a different file system than the one that currently contains **Chapter3** (and **Chapter3.hard**). Note that, after the `mv` command has been executed, the link counts for **Chapter3.hard** and **/tmp/Chapter3** are 1 each and that the files have different inodes; **/tmp/Chapter3** has inode **472** and **Chapter3.hard** has the same old inode **52473**. Note that although the `set` command fails, the `mv` command is successful. The `ln` command cannot link **/tmp/Chapter3** to **temp.hard** because the two files are in different file systems.

```
$ ls -il
total 2
```

```
13059 -rw-r--r-- 1 sarwar faculty  398 Mar 11 14:20 Chapter1
17488 -rw-r--r-- 1 sarwar faculty 5983 Jan 17 11:57 Chapter2
52473 -rw-r--r-- 2 sarwar faculty 9352 May 28 23:09 Chapter3
52473 -rw-r--r-- 2 sarwar faculty 9352 May 28 23:09 Chapter3.hard
$ mv Chapter3 /tmp/Chapter3
mv: /tmp/Chapter3: set owner/group (was: 1004/1008): Operation not
permitted
$ ls -il
total 2
13059 -rw-r--r-- 1 sarwar faculty  398 Mar 11 14:20 Chapter1
17488 -rw-r--r-- 1 sarwar faculty 5983 Jan 17 11:57 Chapter2
52473 -rw-r--r-- 2 sarwar faculty 9352 May 28 23:09 Chapter3.hard
$ ls -il /tmp/Chapter3
472 -rw-r--r-- 1 sarwar root 9352 Aug 23 10:03 /tmp/Chapter3
$ ln /tmp/f3 temp.hard
ln: temp.hard: Cross-device link
$
```

The second problem is that only a superuser can create a hard link to a directory. The `ln` command gives an error message when a non-superuser tries to create a hard link to a directory **myweb**, as in

```
$ ln ~/myweb myweb.hard
ln: /home/sarwar/myweb: Is a directory
$
```

The third problem is that some editors remove the existing version of the file you are editing and put the new versions in new files. When that happens, any hard links to the removed file do not have access to the new file, thereby defeating the purpose of linking (file sharing). Fortunately, none of the commonly used editors do so. Thus, the text editors discussed in Chapter 3 (vim and emacs) are safe to use.

In the following in-chapter exercises, you will use the `ln` and `ls -il` commands to create and identify hard links, and to verify a serious limitation of hard links.

EXERCISE 8.1

Create a file **Ch8Ex1** in your home directory that contains this problem. Establish a hard link to this file, also in your home directory, and call the link **Ch8Ex1.hard**. Verify that the link has been established by using the `ls -il` command. What field in the output of this command did you use for verification?

EXERCISE 8.2

Execute the `ln /tmp ~/tmp` command on your UNIX system. What is the purpose of the command? What happens when you execute the command? Does the result make sense? Why or why not?

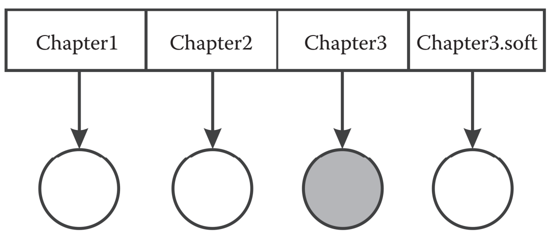
8.6.3 Soft/Symbolic Links

Soft/symbolic links take care of all the problems inherent in hard links and are therefore used more often than hard links. They are different from hard links both conceptually and in terms of how they are implemented. They do have a cost associated with them, which we discuss in Section 8.6.4, but they are extremely flexible and can be used to link files across machines and networks.

You can create soft links by using the `ln` command with the `-s` option. The following session illustrates the creation of a soft link.

```
$ ls -il
total 2
13059 -rw-r--r-- 1 sarwar faculty  398 Mar 11 14:20 Chapter1
17488 -rw-r--r-- 1 sarwar faculty 5983 Jan 17 11:57 Chapter2
52473 -rw-r--r-- 2 sarwar faculty 9352 May 28 23:09 Chapter3
52473 -rw-r--r-- 2 sarwar faculty 9352 May 28 23:09 Chapter3.hard
$ ln -s Chapter3 Chapter3.soft
$ ls -il
total 2
13059 -rw-r--r-- 1 sarwar faculty  398 Mar 11 14:20 Chapter1
17488 -rw-r--r-- 1 sarwar faculty 5983 Jan 17 11:57 Chapter2
52473 -rw-r--r-- 2 sarwar faculty 9352 May 28 23:09 Chapter3
52473 -rw-r--r-- 2 sarwar faculty 9352 May 28 23:09 Chapter3.hard
52479 lrwxr-xr-x 1 sarwar faculty    8 Aug 23 13:57 Chapter3.soft
-> Chapter3
$
```

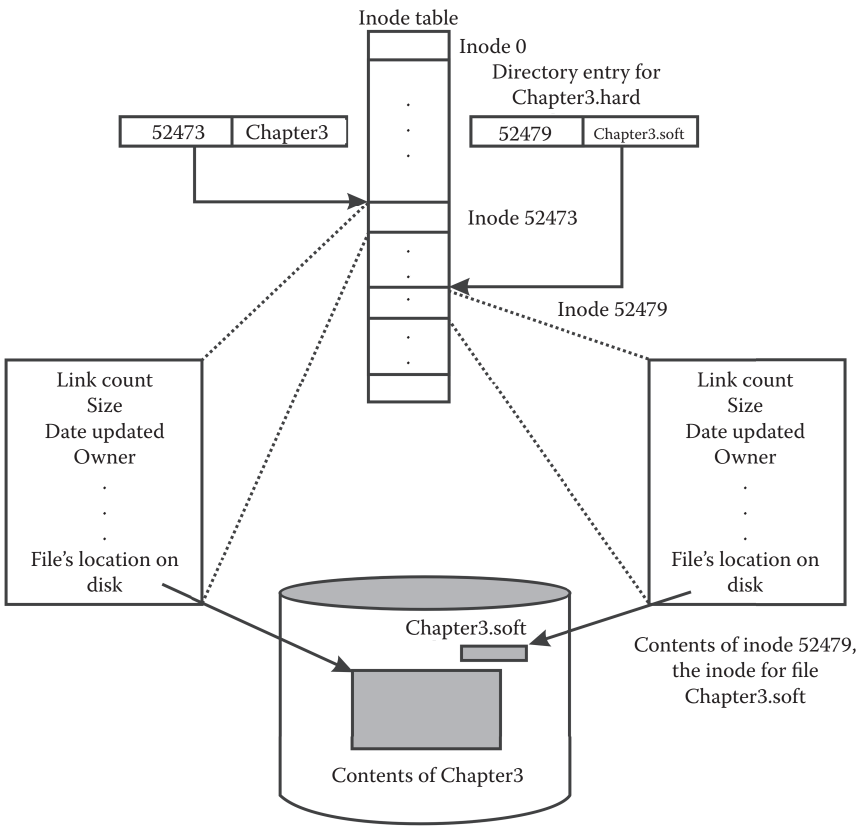
The `ln -s Chapter3 Chapter3.soft` command is used to create a symbolic link to the file **Chapter3** in the present working directory, and the symbolic link is given the name **Chapter3.soft**. The output of the `ls -il` command shows a number of important items that reveal how symbolic links are implemented and how they are identified in the output. First, the original file (**Chapter3**) and the link file (**Chapter3.soft**) have different inode numbers: **52473** for **Chapter3** and **52479** for **Chapter3.soft**, which means that they are different files. Second, the original file (**Chapter3**) is of file type - (ordinary) and the link file (**Chapter3.soft**) is of type `l` (link). Third, the value link count has not changed for **Chapter3** (and **Chapter3.hard**) and is 1 for **Chapter3.soft**, which further indicates that the two files are different. Fourth, the file sizes are different: 9352 bytes for the original file (**Chapter3**) and 8 bytes file the link file (**Chapter3.soft**). Last, the name of the link file is followed by `-> Chapter3`, the pathname for the file that **Chapter3.soft** is a symbolic link to; the string after the -> sign is specified as the first argument in the `ln -s` command. The pathname of the existing file is the content of the link file, which also explains the size of the link file (8 characters in the word `Chapter3`). Figure 8.5 shows the logical file structure of the current directory, directory entries in the current directory, and a diagram that shows that **Chapter3** and **Chapter3.soft** are truly separate files and that the link file contains the pathname of the file to which it is a link.



(a)

Inode #	File
1076	.
2083	..
13059	Chapter1
17488	Chapter2
52473	Chapter3
52479	Chapter3.soft

(b)



(c)

FIGURE 8.5 Establishing a soft link: (a) logical structure of the current directory; (b) contents of the current directory; (c) soft link implementation by establishing a pointer in the link file to (the pathname of) the existing file.

In summary, when you create a symbolic link, a new file of type `l` is created. This file contains the pathname of the existing file as specified in the first argument of the `ln -s` command. When you make a reference to the link file, the UNIX system sees that the type of the file is `l` and reads the link file to find the pathname for the actual file to which you are referring. For example, for the `cat Chapter3.soft` command, the system reads the contents of **Chapter3.soft** to get the name of the file to display (**Chapter3** in this case) and send its contents to standard output. Hence, you see the contents of **Chapter3** displayed.

You can create soft links across file systems. In the following session, we create a symbolic link to the **/bin** directory. The name of the symbolic link is **symlinktobin** and it is placed in the current directory. Note that the inode numbers of **/bin** and **symlinktobin** are different, as expected.

```
$ ln -s /bin symlinktobin
$ ls -ild symlinktobin /bin
  34 drwxr-xr-x 2 root    root     47 Feb 25  2014 /bin
4633 lrwxr-xr-x 1 sarwar  faculty   4 Aug 24 15:24 symlinktobin ->
/bin
$
```

In the following session, we show an example to create a soft link to a file such that the file and its symbolic link reside in different file systems.

```
$ cp Chapter3 /tmp
$ ln -s /tmp/Chapter3 temp.soft
$ ls -il /tmp/Chapter3 temp.soft
  473 -rw-r--r-- 1 sarwar faculty 9352 Aug 23 15:38 /tmp/Chapter3
52497 lrwxr-xr-x 1 sarwar faculty   13 Aug 23 15:38 temp.soft -> /
tmp/Chapter3
$
```

Here, the file **Chapter3** is copied from one file system (that contains this file) to another that contains the **/tmp** directory. Then, the `ln –s /tmp/Chapter3 temp.soft` command is used to create a symbolic link to the copied file. The command works without any problems, establishing a symbolic link to **/tmp/Chapter3** in **temp.soft**. Note that the inode numbers of the two files are different, indicating that the two files are distinct; **temp.soft** contains the pathname of the file for which it is a symbolic link, **/tmp/Chapter3**. Recall that in Section 8.6.1 a similar call to establish a hard link between **/tmp/Chapter3** and **temp.hard** failed.

The following session shows how symbolic links can be created to all the files in a directory, including the directory files. The `ln -sf ~/unixbook/examples/dir1/* ~/unixbook/examples/dir2` command creates soft links to all the files in the directory called **~/unixbook/examples/dir1** and puts them in the directory **~/unixbook/examples/dir2**. You must have execute and write permissions for the **dir2** directory, and execute permission to all the directories in the pathname. The `-f` option is used to force creation of the soft link in case any of the files **f1**, **f2**, or **f3** already exist in **~/unixbook/examples/dir2**. On some systems, the `-f` and `-s` options may not work together, in which case you will use only the `-s` option.

```
$ cd ~/unixbook/examples
$ more dir1/f1
Hello, World!
This is a test file.
$ ls -l dir1
-rw------- 1 sarwar faculty  35 Jun 22 22:21 f1
```

```
-rw------- 1 sarwar faculty 168 Jun 22 22:33 f2
-rw------- 1 sarwar faculty 783 Jun 22 22:35 f3
$ ln -sf ~/unixbook/examples/dir1/* ~/unixbook/examples/dir2
$ ls -l dir2
lrwxr-xr-x 1 sarwar faculty 38 Jun 22 22:54 f1 -> /home/sarwar/
unixbook/examples/dir1/f1
lrwxr-xr-x 1 sarwar faculty 38 Jun 22 22:54 f2 -> /home/sarwar/
unixbook/examples/dir1/f2
lrwxr-xr-x 1 sarwar faculty 38 Jun 22 22:54 f3 -> /home/sarwar/
unixbook/examples/dir1/f3
$ more dir2/f1
Hello, World!
This is a test file.
$
```

You can run the following command to create a symbolic link in your home directory to the file **/home/sarwar/unixbook/examples/demo1**. The soft link appears as a file called **demo1** in your home directory. If **demo1** already exists in your home directory, you can overwrite it with the `-f` option. If **demo1** exists in the home directory and you don't use the `-f` option, an error message is displayed on the screen informing you that the **demo1** file exists. You must have the execute permission for the directories in the pathname **/home/sarwar/unixbook/examples/demo1**, and **demo1** must be a file.

```
$ ln -sf /home/sarwar/unixbook/examples/demo1 ~
$
```

The user **sarwar** can run the following command to create a soft link called **demo1** in a directory **dir1** in **bob**'s home directory that points to the **/home/sarwar/unixbook/examples/demo1** file. Figure 8.6 shows how the soft link is established.

```
$ ln -sf /home/sarwar/unixbook/examples/demo1 /home/bob/dir1
$
```

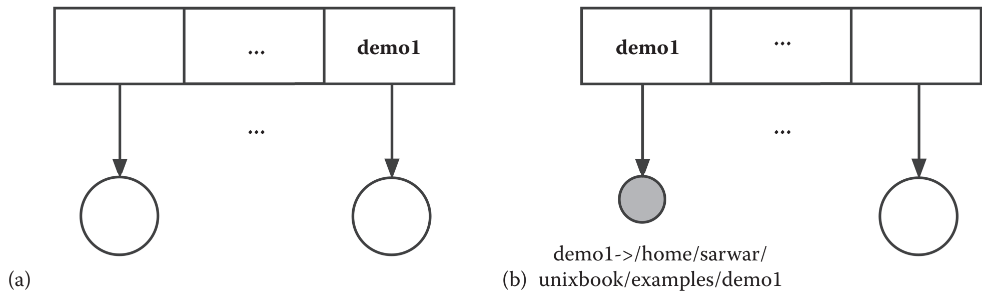


FIGURE 8.6 A soft link between (a) **/home/sarwar/unixbook/examples/demo1** and (b) **/home/bob/dir1**.

The user **sarwar** must have execute permission for **bob**'s home directory, and execute and write permission for **dir1** (the directory in which the soft link is created). The user **bob** must have proper access permissions for **demo1** in **sarwar**'s directory structure to access this file. Thus, if **sarwar** and **bob** are in the same user group and **bob** has to edit **memo1**, then **sarwar** must set the group access privileges on the file to read and write. The user **bob** can then edit **demo1** by using, for example, the `vi demo1` command from his home directory.

The following command accomplishes the same task. Remember that **sarwar** runs this command.

```
$ ln -sf ~/unixbook/examples/demo1 /home/bob/dir1
$
```

You can run the following `ln` command to create soft links to all the files, including directory files, in your **~/unixbook/examples** directory. These soft links reside in the directory called **unixbook/examples** in **john**'s home directory and have the names of the original files. The user **john** must create the **unixbook** directory in his **home** directory and the **examples** directory in his **unixbook** directory. You must have execute permission for **john**'s **unixbook** directory and execute and write permission for his **examples** directory in order for the command to run successfully.

```
$ ln -sf ~/unixbook/examples/* /home/john/unixbook/examples
$
```

8.6.4 Pros and Cons of Symbolic Links

As previously stated, symbolic links do not have the problems and limitations of hard links. Thus, symbolic links can be established to directories and between files across file systems. Also, files that symbolic links point to can be edited by any kind of editor without any ill effects, provided that the file's pathname doesn't get changed—that is, the original file is not moved.

Symbolic links do have a problem of their own that is not associated with hard links: if the file that the symbolic link points to is moved from one directory to another, it can no longer be accessed via the link. The reason is that the link file contains the pathname for the original location of the file in the file structure. When the file location is changed, the link becomes *dangling*; that is, it points to a file that does not exist at the specified (original) location. You also have a dangling pointer if the original file is deleted. The following session illustrates this point.

```
$ mv /tmp/Chapter3 .
$ cat temp.soft
cat: temp.soft: No such file or directory
$
```

Suppose that **temp.soft** is a symbolic link to the file **/tmp/Chapter3**. The `mv` command is used to move **/tmp/Chapter3** to the present working directory. The `cat` command fails because the soft link still points to the file with pathname **/tmp/Chapter3**. This result is quite logical but is still a drawback; in hard links, the `cat` command would not fail so long as the moved file stays within the same file system.

Some other drawbacks of the symbolic links are that UNIX has to support an additional file type (the link type) and a new file has to be created for every link. Creation of the link file results in space overhead for an extra inode, disk space needed to store the pathname of the file to which it is a link, and other kernel data structures. Symbolic links also result in slower file operations because, for every reference to the file, the link file has to be opened and read in order for you to reach the actual file. The actual file is then processed for reading or writing, requiring an extra disk access to be performed if a file is referenced via a symbolic link to the file.

In the following in-chapter exercises, you will use the `ln -s` and `ls -il` commands to create and identify soft links, and to verify that you can create soft links across file systems.

EXERCISE 8.3

Establish a soft link to the file **Ch8Ex1** that you created in Exercise 8.1. Call the soft link **Ch8Ex1.soft**. Verify that the link has been established. What commands did you use to establish the link and verify its creation?

EXERCISE 8.4

Execute the `ln -s /tmp ~/tmp` command on your UNIX system. What is the purpose of the command? What happens when you execute the command? Does the result make sense? Why or why not?

SUMMARY

Any of several techniques can be used to allow a team of users to share UNIX files and directories. Some of the most commonly used methods of file sharing are duplicating the files to be shared and distributing them among team members, establishing a common account for team members, setting appropriate permissions on the files to be shared, setting up a UNIX user group for the team members, and establishing links to the shared files in the directories of all team members. File sharing via hard and soft links is the main topic of this chapter. However, the issue of simultaneous access of shared files by team members is not discussed here (see Chapter 17).

Hard links allow you to refer to an existing file by another name. Although hard links are the primary mechanism used by UNIX to glue the file system structure, they have several shortcomings. First, an existing file and its links must be in the same file system. Second, only a superuser can create hard links to directories. Third, moving a file to another file system breaks all links to it.

Soft links can be used to overcome the problems associated with hard links. When a soft link to a file is created, a new file is created that contains the pathname of the file to which it is a link. The type of this file is link. Soft links are expensive in terms of the time needed to access the file and the space overhead of the link file. The time overhead during file access occurs because the link file has to be opened in order for the pathname of the actual file to be read (or written to, whatever the case may be), and only then does the actual process of file opening and reading (or writing) take place. The link file that contains the pathname of the original file causes the space overhead.

Hard and soft links are established with the `ln` command. For creating soft links, the `-s` option is used with the command. The `ls -il` command is used to identify (or confirm establishment of) links. The first field of the output of this command identifies the inode numbers for the files in a directory, and all hard links to a file have the same inode number as the original file. The first letter of the second field represents file type (`l` for *soft link*) and the remaining letters specify file permissions. The third field identifies the number of hard links to a file. Every simple file has one hard link at the time it is created. The last field identifies file names; a soft link's name is followed by -> filename, where filename is the name of the original file. The `-f` option can be used to force the creation of a link—that is, to overwrite an existing file with the newly created link.

QUESTIONS AND PROBLEMS

1. What are the five methods that can be used to allow a team of users to share files in UNIX?
2. What is a link in UNIX? Name the types of link that UNIX supports. How do they differ from each other?
3. What are the problems with hard links?
4. Remove the file **Ch8Ex1** that you created in Exercise 8.1. Display the contents of **Ch8Ex1.hard** and **Ch8Ex1.soft**. What happens? What command did you use for displaying the files? Does the result make sense? Why or why not?
5. Search the **/usr/bin** directory on your system and identify three links in it. Write down the names of these links. Are these hard or soft links? How do you know?
6. While in your home directory, can you establish a hard and soft link to **/etc/passwd** on your system? Why? What commands did you use? Are you satisfied with the results of the command execution?
7. Every UNIX directory has at least two hard links. Why?
8. Can you find the number of hard and soft links to a file? If so, what command(s) do you need to use?
9. Suppose that a file called **shared** in your present directory has five hard links to it. Give a sequence of commands to display the absolute pathnames of all of these links. (*Hint*: Use the `find` command.)

10. Create a directory, **dir1**, in your home directory and three files, **f1**, **f2**, and **f3**, in it. Ask a friend to create a directory, **dir2**, in his or her home directory, with **dir1.hard** and **dir1.soft** as its subdirectories. Create hard and soft links to all the files in your **dir1** in your friend's **~/dir2/dir1.hard** and **~/dir2/dir1.soft** directories. Give the sequence of commands that you executed to do so.
11. For Problem 10, what are the inode numbers of the hard links and soft links? What command did you use to determine them? What are the contents of the link (both hard and soft) files? How did you get your answers?
12. What are the pros and cons of symbolic links?
13. Clearly describe how file sharing can be accomplished by using links (hard and soft) in UNIX. In particular, do you need to do anything other than establish links to the files to be shared?
14. Suppose you have a collection of data files, say **file1.data**, **…**, **file9.data**, that need to be shared (read only) among 100 programs in your group. Discuss the overhead involved for each of the following:
    a. Setting permissions
    b. Creating hard links
    c. Creating soft links
    d. Making individual private copies of each file
15. Browse through the root (/) directory and its subdirectories. Identify 10 soft links and write them in a table, along with the name of each link and the directory in which it is found.
16. Symbolic links have the *dangling pointer* problem. What is it? Explain with an example.
17. Hard links may not be established between files across file systems. What is the technical reason for this limitation of hard links? (*Hint*: Think about inodes and file systems.)

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

CHAPTER 9

Redirection and Piping

Objectives

- To describe the notion of standard files—standard input, standard output, and standard error files—and file descriptors
- To describe input and output redirection for standard files
- To discuss the concept of error redirection and appending to a file
- To explain the concept of pipes in UNIX
- To describe how powerful operations can be performed by combining pipes, file descriptors, and redirection operators
- To discuss error redirection in the C shell
- To explain the concept of FIFOs (also known as named pipes) and their command line use
- To cover the commands and primitives

```
&, |, <, >, >>, cat, diff, grep, lp, mkfifo, more, pr, sort,
stderr, stdin, stdout, tee, tr, uniq, wc
```

9.1 INTRODUCTION

All computer software (commands) performs one or more of the following operations: input, processing, and output; a typical command performs all three. The question for the operating system is: Where does a shell command (internal or external) take its input from, where does it send its output to, and where are the error messages sent to? If the input to a command is not part of the command code (i.e., data within the code in the form of constants and/or variables), it must come from an outside source. This outside source is usually a file, although it could be an input/output (I/O) device such as a keyboard or a network interface card. Command output and error messages can go to a file as well. For a command to read from or write to a file, it must first open the file.

There are default files where a command reads its input and sends its output and error messages, called *standard input*, *standard output*, and *standard error*. In UNIX, these files are known as *standard files* for a command. The input, output, and errors of a command can be redirected to other files by using *file redirection facilities* in UNIX. This allows you to connect several commands together to perform a complex task that cannot be performed by a single existing command. We discuss the notion of standard files and redirection of input, output, and error in UNIX in this chapter.

9.2 STANDARD FILES

In UNIX, three files are automatically opened by the kernel for every command to read input from and send its output and error messages to. These files are known as standard files: standard input (**stdin**), standard output (**stdout**), and standard error (**stderr**). By default, these files are associated with the terminal on which the command executes. More specifically, the keyboard is standard input, and the display screen (or the console at which you are logged in) is standard output and standard error. Therefore, every command, by default, takes its input from the keyboard and sends its output and error messages to the display screen (or the console window), as shown in Figure 9.1. Recall our explanation of the per-process file descriptor table in Chapter 4. In the remainder of this chapter, we use the terms *monitor screen*, *display screen*, *console window*, and *display window* interchangeably.

9.3 INPUT REDIRECTION

Input redirection is accomplished by using the less-than symbol (<). The following syntax is used to detach the keyboard from the standard input of `command` and attach **input-file** to it. Thus, if `command` reads its input from standard input, this input will come from **input-file**, not the keyboard attached to the terminal on which the command is run. The

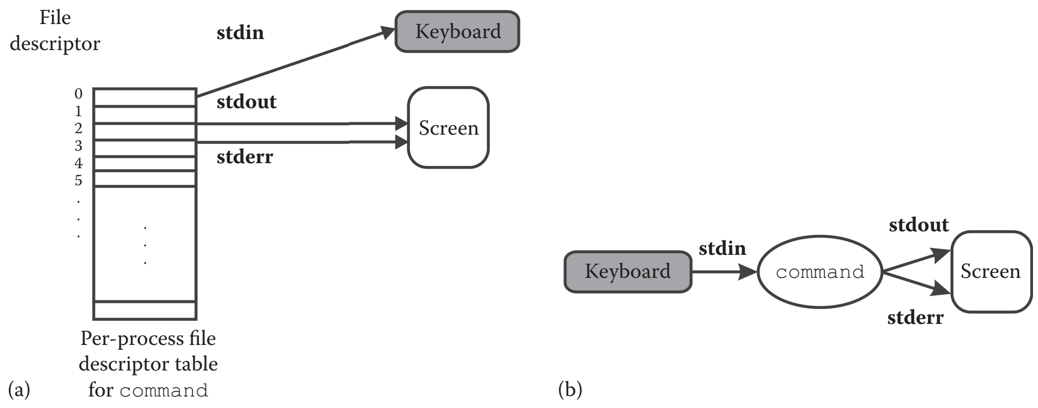


FIGURE 9.1 Standard files and file descriptors: (a) file descriptors; (b) semantics of a command execution.

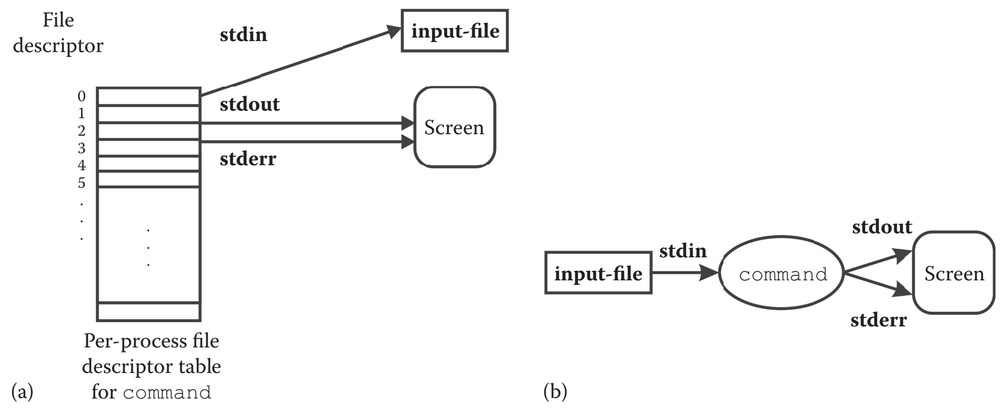


FIGURE 9.2 Input redirection: (a) file descriptors and standard files for `command`; (b) semantics of input redirection.

semantics of the command syntax are shown in Figure 9.2. Note that the `command` input comes from **input-file**.

SYNTAX

```
command < input-file
```

Purpose: Input to `command` comes from **input-file** instead of from the keyboard

For example, the command `cat < tempfile` reads input from **tempfile** (as opposed to the keyboard, because the standard input for `cat` has been attached to **tempfile**) and sends its output to the display screen. So, effectively, the contents of **tempfile** are displayed on the monitor screen. This command is different from `cat  tempfile`, in which **tempfile** is passed as a command line argument to the `cat` command; the standard input of `cat` does not change and is still the keyboard attached to the terminal on which the command is run.

Similarly, in the command `grep "John" < Phones`, the `grep` command reads its input from the **Phones** file in the current directory, not from the keyboard. The output and error messages of the command go to the display screen. Again, this command is different from `grep "John" Phones`, in which the **Phones** file is passed as an argument to `grep`; the standard input of `grep` does not change and is still the keyboard attached to the terminal on which the command executes. However, the net effect of the `grep` command is the same in both cases from a user's perspective. Similarly, the use of < is not needed in most cases because the command reads from a file in any case.

The `cat` and `grep` commands take input from standard input if they are not passed file arguments from the command line. The `tr` command takes input from standard input only and sends its output to standard output. The command does not work with a file as a command line argument. Thus, input redirection is often used with the `tr` command, as in `tr -s ''''< Bigfile`. When this command is executed, it substitutes multiple spaces in **Bigfile** with single spaces.

9.4 OUTPUT REDIRECTION

Output redirection is accomplished by using the greater-than symbol (>). The following syntax is used to detach the display screen from the standard output of `command` and attach **output-file** to it. Thus, if `command` writes/sends its output to standard output, the output goes to **output-file**, not the monitor screen attached to the terminal on which the command runs. The error messages still go to the display screen, as before. The semantics of the command syntax are shown in Figure 9.3.

SYNTAX

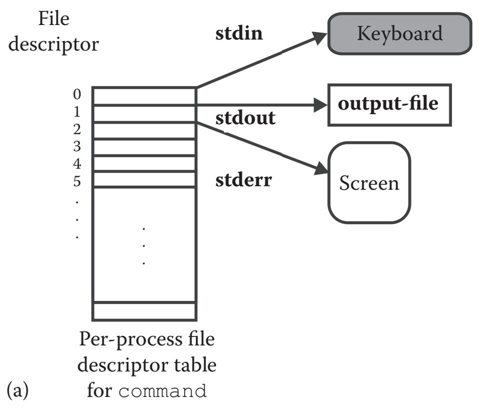
```
command > output-file
```

Purpose: Send output of `command` to the file **output-file** instead of to the monitor screen

Consider the `cat > newfile` command. Recall that the `cat` command sends its output to standard output, which is the display screen by default. This command syntax detaches the display screen from standard output of the `cat` command and attaches **newfile** to it. The standard input of `cat` remains attached to the keyboard. When this command is executed, it creates a file called **newfile** whose contents are whatever you type on the keyboard until you hit `<Ctrl-D>` in the first column of a new line. If **newfile** already exists, by default it is overwritten.

Similarly, the command `grep "John" Phones > Phone_John` sends its output (lines in the **Phones** file that contain the word "John") to a file called **Phone_John**, as opposed to displaying it on the monitor screen. The input for the command comes from the **Phones** file. The command terminates when `grep` encounters the end-of-file (eof) character in **Phones**.

In a network environment, the following command can be used to sort the file **datafile** residing on the computer that you are currently logged on to (the client computer), on



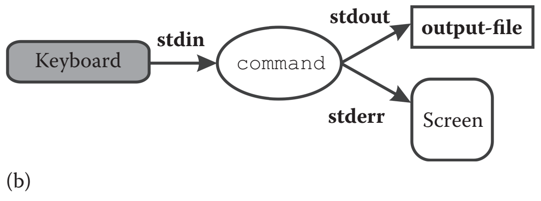


FIGURE 9.3 Output redirection: (a) file descriptors and standard files for `command`; (b) semantics of output redirection.

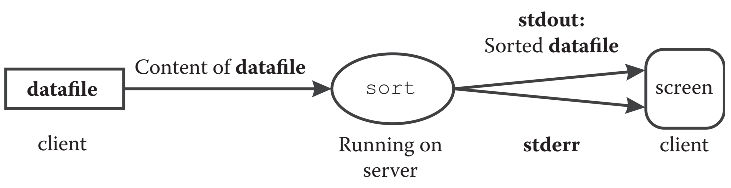


FIGURE 9.4 Semantics of the `ssh server sort < datafile` command run on **mymachine**.

the computer called **server**. The output of the command—that is, the sorted data—is sent to the display screen of the client computer. Figure 9.4 illustrates the semantics of this command.

```
$ ssh server sort < datafile
$
```

This command is a good example of how multiple computers can be used to perform various tasks concurrently in a network environment. It is a useful command if your computer (call it **client**) has a large file, **datafile**, to be sorted and you do not want to make multiple copies of the file on various computers on the network to prevent inconsistency in them. This command allows you to perform the task. Such commands also are useful if the server has specialized UNIX tools that you are allowed to use but not allowed to make copies of on your machine. We discuss network-related UNIX commands and utilities in Chapter 11. We have used this example to illustrate the power of the UNIX I/O redirection feature, not to digress to computing in a network environment.

The following session shows the contents of the **Students** file on the local (client) machine and the output of the `sort` command executed on a remote (server) machine 198.102.10.20 under **sarwar**'s login. Since **sarwar**'s login is password protected, the system prompted him for a password before running the `sort` command on the remote computer and displaying its output on the local computer. Note that we have shown the Internet Protocol (IP) address of a fictitious `ssh` server, but the session was executed on a real remote machine.

```
$ cat Students
John Doe    ECE   3.54  A
Pam Meyer   CS    3.61  A
Jim Davis   CS    2.71  B
John Doe    ECE   3.54  A
Jason Kim   ECE   3.97  A
Amy Nash    ECE   2.38  C
$ ssh sarwar@198.102.10.20 sort < Students
Password for sarwar@pcbsd-srv:
Amy Nash    ECE   2.38  C
Jason Kim   ECE   3.97  A
Jim Davis   CS    2.71  B
```

```
John Doe     ECE    3.54   A
John Doe     ECE    3.54   A
Pam Meyer    CS     3.61   A
$
```

9.5 COMBINING INPUT AND OUTPUT REDIRECTION

Input and output redirections can be used together, according to the syntax given in the following command description.

SYNTAX

```
command < input-file > output-file
command > output-file < input-file
```

Purpose: Input to `command` comes from **input-file** instead of the keyboard and output of `command` goes to **output-file** instead of the display screen

When this syntax is used, `command` takes its input from **input-file** (not from the keyboard attached to the terminal) and sends its output to **output-file** (not to the display screen), as shown in Figure 9.5.

In the `cat <lab1 > lab2` command, the `cat` command takes its input from the **lab1** file and sends its output to the **lab2** file. The net effect of this command is that a copy of **lab1** is created in **lab2**. Therefore, this command line is equivalent to `cp lab1 lab2`, provided that **lab2** does not exist. If **lab2** is an existing file, the two commands have different semantics. The `cat <lab1 > lab2` command truncates **lab2** (sets its size to zero and the read/write pointer to the first byte position) and overwrites it with the contents of **lab1**. Because **lab2** is not recreated, its attributes (e.g., access permissions and link count) are not changed. In the case of the `cp lab1 lab2` command, not only is the data in **lab1** copied into **lab2**, but also its attributes from its inode are copied into the inode for **lab2**. Thus, the `cp` command results in a true copy (data and attributes) of **lab1** into **lab2**.

In the following in-chapter exercises, you will practice the use of input and output direction features of UNIX.

EXERCISE 9.1

Write a shell command that counts the number of characters, words, and lines in a file called **memo** in your present working directory and shows these values on the display screen. Use input redirection.

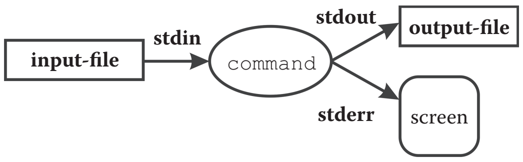


FIGURE 9.5 Combined use of input and output redirection.

EXERCISE 9.2

Repeat Exercise 9.1, but redirect output to a file called **counts.memo**. Use I/O redirection.

9.6 I/O REDIRECTION WITH FILE DESCRIPTORS

As described in Section 4.6, the UNIX kernel associates a small integer number with every open file, called the *file descriptor* for the file. The file descriptors for standard input, standard output, and standard error are 0, 1, and 2, respectively. The Bourne, Korn, Bash, and POSIX shells allow you to open files and associate file descriptors with them; the C shell does not allow the use of file descriptors. The other descriptors, usually ranging from 3 onward, are used when a process opens files simultaneously. These descriptors are called *user-defined file descriptors*. The upper limit of these descriptors (and hence the number of files that a process may open simultaneously) varies from system to system and may be determined by running the `uname –n` command. Each descriptor is used to index a kernel table, called the *per-process file descriptor table*, as discussed briefly in Section 4.7. A more detailed discussion on this topic may be found in Chapters 18 through 21. In the following sections, we describe I/O and error redirection under the Bourne, Korn, Bash, and POSIX shells. We discuss the C shell syntaxes and give examples in Section 9.13.

By making use of file descriptors, standard input and standard output can be redirected in the Bourne, Korn, Bash, and POSIX shells by using the `0<` and `1>` operators, respectively. Therefore, `cat 1> outfile`, which is equivalent to `cat > outfile`, takes input from standard input and sends it to **outfile**; error messages go to the display screen. Similarly, `ls -l foo 1> outfile` is equivalent to `ls -l foo > outfile`. The output of this command (the long listing for **foo**) goes into a file called **outfile**, and error messages generated by it go to the display screen.

The file descriptor `0` can be used as a prefix with the < operator to explicitly specify input redirection from a file. In the command shown below, the input to the `grep` command is the contents of **tempfile** in the present working directory.

```
$ grep "John" 0< tempfile
... command output ...
$
```

9.7 REDIRECTING STANDARD ERROR

The standard error of a command can be redirected by using the `2>` operator (i.e., associating the file descriptor for standard error with the > operator) as follows.

SYNTAX

```
command 2> error-file
```

Purpose: Error messages generated by `command` and, by default, sent to **stderr** are redirected to **error-file**

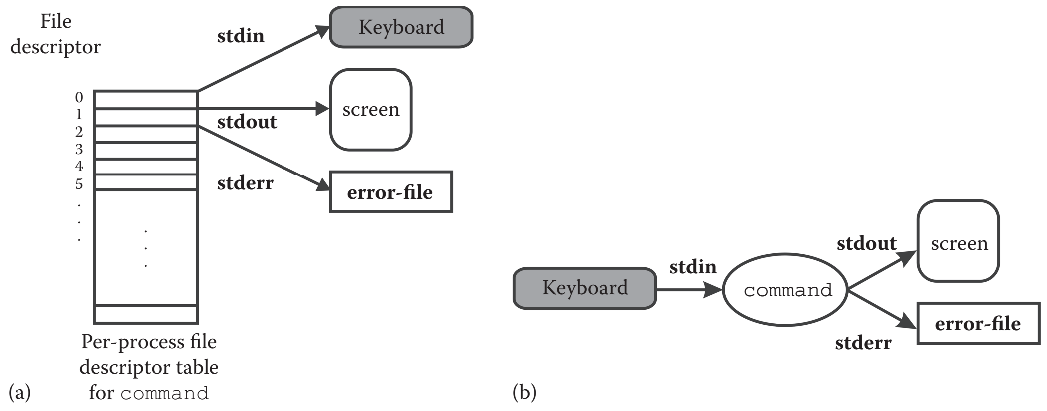


FIGURE 9.6 Error redirection: (a) standard descriptors and standard files for `command`; (b) semantics of error redirection.

With this syntax, `command` takes its input from the keyboard, sends its output to the monitor screen, and any error messages produced by `command` are sent to **error-file**. The semantics of the command syntax are shown in Figure 9.6. Command input may come from a file passed as a command line argument.

The command `grep "John" Phones 2> error.log` takes input from the **Phones** file, sends output to the display screen, and any error message produced by `grep` goes to a file called **error.log**. If **error.log** already exists, it is overwritten; otherwise, it is created. The following example shows how the standard error of `ls -l` can be redirected to a file.

```
$ ls -l foo 2> error.log
... long listing for foo if no errors ...
$
```

The output of `ls -l foo` goes to the display screen, and error messages go to **error.log**. Thus, if **foo** does not exist, the error message `ls: foo: No such file or directory` goes into the **error.log** file, as shown in the following session. The actual wording of the message varies from system to system, but it basically informs you that **foo** does not exist.

```
$ ls -l foo 2> error.log
$ cat error.log
ls: foo: No such file or directory
$
```

Keeping standard error attached to the display screen and not redirecting it to a file is useful in many situations. For example, when the `cat lab1 lab2 lab3 > all` command is executed to concatenate files **lab1**, **lab2**, and **lab3** into a file called **all**, you would want to know whether any of the three input files are nonexistent or if you do not have permission to read any of them. In this case, redirecting the error message to a file does not make much sense because you want to see the immediate results of the command execution.

9.8 REDIRECTING **stdout** AND **stderr** IN ONE COMMAND

Standard output and standard error can be redirected to the same file. One obvious way to do so is to redirect **stdout** and **stderr** to the same file by using file descriptors with the > symbol, as in the following command.

```
$ cat lab1 lab2 lab3 1> cat.output 2> cat.errors
$
```

In this case, the input of the cat command comes from the **lab1**, **lab2**, and **lab3** files, its output goes to the **cat.output** file, and any error message goes to the **cat.errors** file, as shown in Figure 9.7. Note that, although not shown in Figure 9.7b, files **lab1**, **lab2**, and **lab3** have file descriptors assigned to them when they are opened for reading by the cat command. The command produces an error message if any one of the three "lab" files does not exist or if you do not have read permission for any of these files.

The following command redirects the **stdout** and **stderr** of the cat command to the **cat.output.errors** file. Thus, the same file (**cat.output.errors**) contains the output of the cat command, along with any error messages that may be produced by the command.

```
$ cat lab1 lab2 lab3 1> cat.output.errors 2>&1
$
```

In this command, the string 2>&1 tells the command shell to make descriptor 2 a duplicate of descriptor 1, resulting in the error messages going to the same place that the command output goes to. Similarly, the string 1>&2 can be used to tell the command shell to make descriptor 1 a duplicate of descriptor 2. Thus, the following command accomplishes the same task. Figure 9.8 shows the semantics of the two commands.

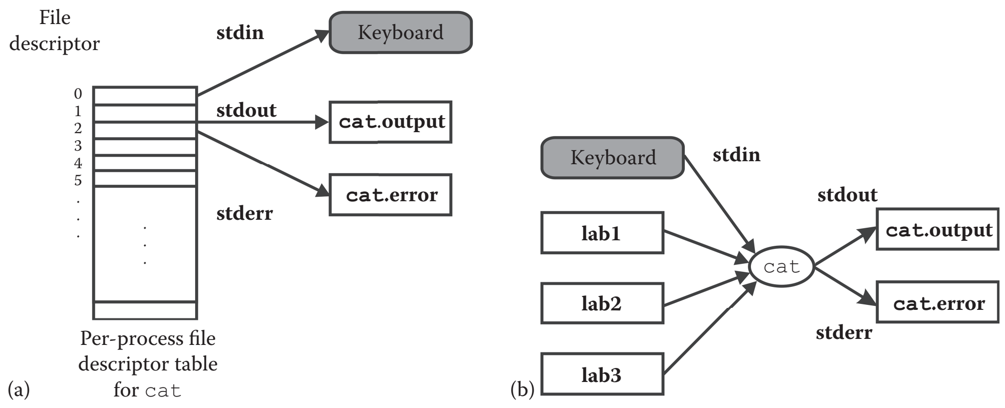


FIGURE 9.7 Error redirection: (a) file descriptors and standard files for cat lab1 lab2 lab3 1> cat.output 2> cat.errors; (b) semantics of the cat command.

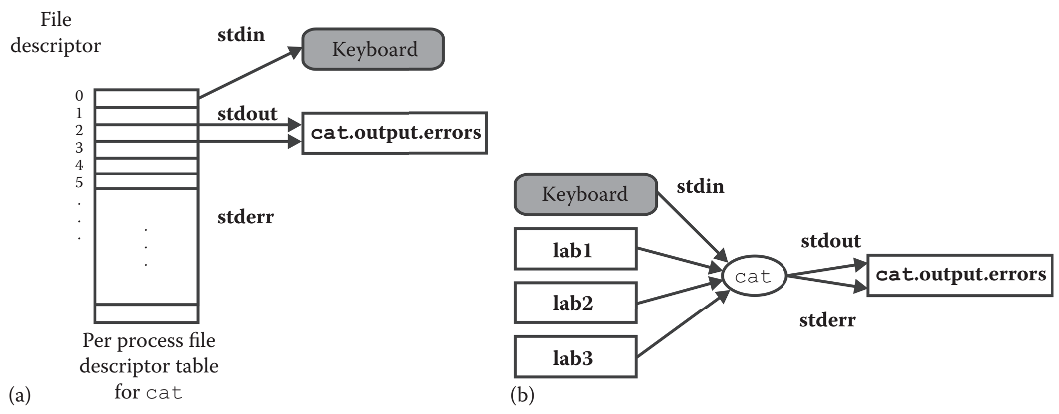


FIGURE 9.8 Error redirection: (a) file descriptors and standard files; (b) semantics of the `cat lab1 lab2 lab3 1> cat.output.errors 2>&1` and `cat lab1 lab2 lab3 2> cat.output.errors 1>&2` commands.

```
$ cat lab1 lab2 lab3 2> cat.output.errors 1>&2
$
```

The evaluation of the command line content for file redirection is left to right. Therefore, redirections must be specified in the left-to-right order if one notation is dependent on another. In the preceding command, first, **stderr** is changed to the file **cat.output.errors**, and then **stdout** becomes a duplicate of **stderr**. Thus, the output and errors for the command both go to the same file, **cat.output.errors**.

The following command therefore does *not* have the effect of the two commands just discussed. The reason is that, in this command, **stderr** is made a duplicate of **stdout** *before* output redirection. Therefore, **stderr** becomes a duplicate of **stdout** (the display screen at this time) first, and then **stdout** is changed to the file **cat.output.errors**. Thus, the output of the command goes to **cat.output.errors** and errors go to the display screen. The sequence shown in Figure 9.9 illustrates the semantics of this command.

```
$ cat lab1 lab2 lab3 2>&1 1> cat.output.errors
$
```

Note that Figure 9.9a and b are identical because the execution of `cat lab1 lab2 lab3 2>&1` does not make any changes to **stdout** and **stderr**—they stay attached to the display screen before and after the command is executed.

9.9 REDIRECTING **stdin**, **stdout**, AND **STDERR** IN ONE COMMAND

Standard input, standard output, and standard error can be redirected in a single command according to the following syntax.

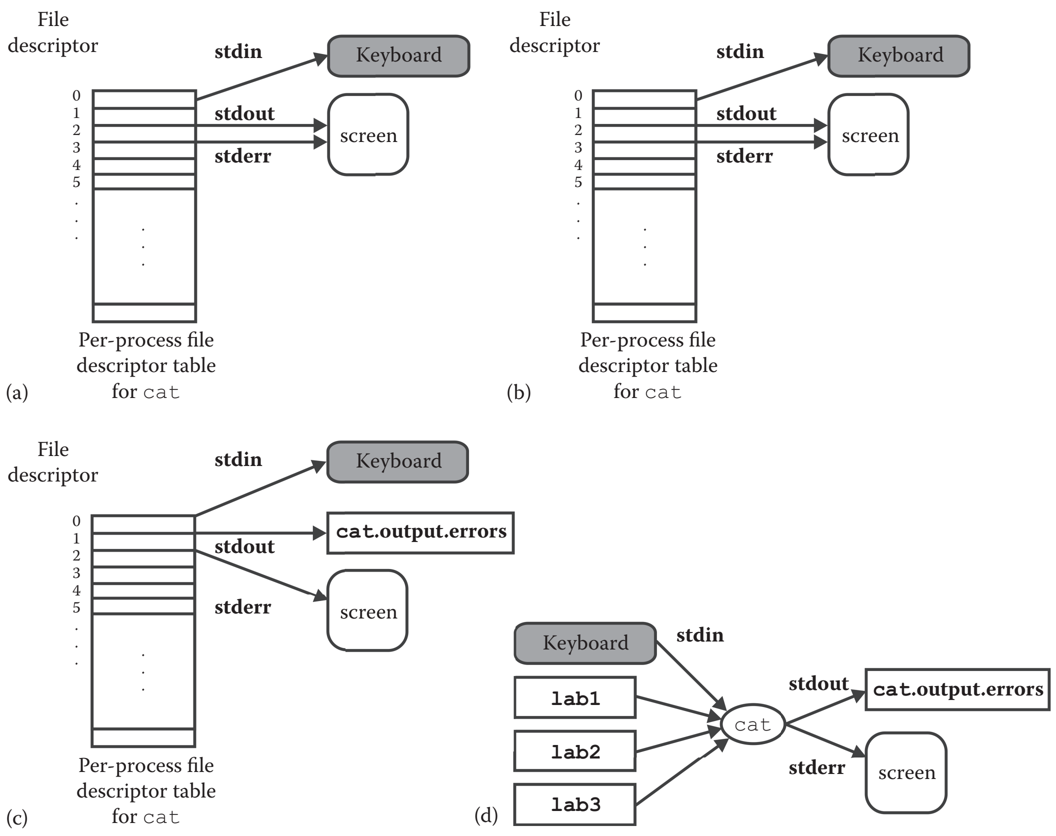


FIGURE 9.9 Output and error redirection: (a) file descriptors and standard files for the `cat` command; (b) standard files after `cat lab1 lab2 lab3 2>&1` with no change in **stdout** and **stderr**; (c) standard files after `cat lab1 lab2 lab3 2>&1 1> cat.output.errors`; and (d) command semantics.

SYNTAX

```
command 0< input-file 1> output-file 2> error-file
```

Purpose: Input to `command` comes from **input-file** instead of the keyboard, output of `command` goes to **output-file** instead of the display screen, and error messages generated by `command` are sent to **error-file** instead of the display screen

The file descriptors `0` and `1` are not required because they are the default values for the < and > operators, respectively. The semantics of this command syntax are shown in Figure 9.10. Evaluation of the command line content for file redirection is left to right, so the order of redirection is important. Consider the following command syntaxes. For the first command, if **input-file** is not found, the error message is sent to the display screen, because **stderr** has not been redirected yet. For the second command, if **input-file** is not found, the error message goes to **error-file** because **stderr** has been redirected to this file.

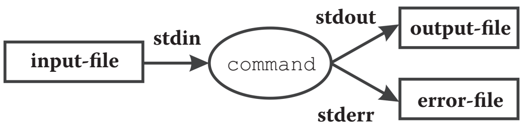


FIGURE 9.10 Redirecting **stdin**, **stdout**, and **stderr** in a single command.

If **error-file** exists, the outputs of the first and second commands go to **stdout** and **output-file**, respectively.

```
command 1> output-file 0< input-file 2> error-file
command 2> error-file 1> output-file 0< input-file
```

The following `sort` command sorts lines in a file called **students** and stores the sorted file in **students.sorted**. If the `sort` command fails to start because the **students** file does not exist, the error message goes to the display screen as shown, not to the file **sort.error**. The reason is that, at the time the shell determines that the **students** file does not exist, **stderr** is still attached to the console.

```
$ sort 0< students 1> students.sorted 2> sort.error
cannot open students: No such file or directory
$
```

For the following command, the error message goes to the **sort.error** file if the `sort` command fails because the **students** file does not exist. The reason is that the shell processes error redirection before it determines that the **students** file is nonexistent.

```
$ sort 2> sort.error 0< students 1> students.sorted
$ cat sort.error
cannot open students: No such file or directory
$
```

9.10 REDIRECTING WITHOUT OVERWRITING FILE CONTENTS (APPENDING)

By default, output and error redirections overwrite contents of the destination file. To *append* output or errors generated by a command to the end of a file, replace the > operator with the >> operator. The default file descriptor with >> is `1`, but file descriptor `2` can be used to append errors to a file. In the following command, the output of `ls -l` is appended to the **output.dat** file, and the error messages are appended to the **error.log** file.

```
$ ls -l 1>> output.dat 2>> error.log
$
```

The following command appends the contents of the files **memo** and **letter** to the end of the file **stuff**. If the command produces any error message, it goes to the **error.log** file. If **error.log** is an existing file, its contents are overwritten with the error message.

```
$ cat memo letter >> stuff 2> error.log
$
```

If you want to keep the existing contents of **error.log** and append new error messages to it, use the following command. For this command, the previous contents of **error.log** are appended with any error message produced by the `cat` command.

```
$ cat memo letter >> stuff 2>> error.log
$
```

The Bourne shell, by default, overwrites a file when the **stdout** or **stderr** of a command is redirected to it, but the Korn, C, and Bash shells have a `noclobber` option that prevents you from overwriting important files accidentally. We discuss this option for the C shell in Section 9.13, but discuss it for the Korn and Bash shells here.

You can set the `noclobber` option in the Korn and Bash shells by using the `set` command with the `-o` option as shown. Of course, if you want to set this option permanently, put the command in your **~/.profile** file.

```
$ set -o noclobber
$
```

In the Bash shell, you can also set the option by using the `set -C` command. When you set the `noclobber` option, you can force overwriting of a file by using the >| operator.

In the following in-chapter exercises, you will practice the use of input, output, and error redirection features of UNIX shells (excluding the C shell) in a command line.

EXERCISE 9.3

Write a command that counts the number of characters, words, and lines in a file called **memo** in your present working directory and writes these values into a file **memo.size**. If the command fails, the error message should go to a file **error.log**. Use I/O and error redirections.

EXERCISE 9.4

Write a shell command to send the contents of a file **greetings** to **doe@domain.com** by using the `mail` command. If the `mail` command fails, the error message should go to a file **mail.errors**. Use input and error redirection.

EXERCISE 9.5

Repeat Exercise 9.2, but append error messages at the end of **mail.errors**.

9.11 UNIX PIPES

The UNIX system allows **stdout** of a command to be connected to the **stdin** of another command. You can make it do so by using the *pipe character* (`|`) according to the following syntax.

SYNTAX

```
command1 | command2 | command3 | ... | commandN
```

Purpose: The standard output of `command1` is connected to the **stdin** of `command2`, the **stdout** of `command2` is connected to the **stdin** of `command3`, ..., and the **stdout** of commandN−1 is connected to the **stdin** of `commandN`

Figure 9.11 illustrates the semantics of this command.

Thus, a pipe allows you to send output of a command as input to another command. The commands that are connected via a pipe are called *filters*. A filter belongs to a class of UNIX commands that take input from **stdin**, manipulate it in some specific fashion, and send it to **stdout**. Pipes and filters are frequently used in UNIX to perform complicated tasks that cannot be performed with a single command. Some commonly used filters are `cat`, `compress`, `crypt`, `grep`, `less`, `lp`, `more`, `pr`, `sort`, `tr`, `uniq`, and `wc`. If a command at the output of a pipe processes its input at a rate slower than the command connected to its input, the pipe stores the excess data and serves it to the command at the output on a first-in-first-out basis. Thus, if `cmd2` in Figure 9.11 processes the incoming data at a slower rate than the rate at which `cmd1` produces data, the excess data produced by `cmd1` is stored in the pipe between `cmd1` and `cmd2`, which serves it to `cmd2` on a first-in-first-out basis. The maximum data that a pipe can store in PC-BSD is dictated by the symbolic constant _POSIX_PIPE_BUF in **/usr/include/limits.h**.

For example, in `ls -l | more`, the `more` command takes the output of `ls -l` as its input. The net effect of this command is that the output of `ls -l` is displayed one screen at a time. The pipe really acts like a water pipe, taking the output of `ls -l` and giving it to `more` as its input, as shown in Figure 9.12.

This command does not use a disk to connect the standard output of `ls -l` to the standard input of `more`, because the pipe is implemented in the kernel area of the main memory. In terms of the I/O redirection operators, the command is equivalent to the following sequence of commands.

```
$ ls -l > temp
$ more < temp (or more temp)
```

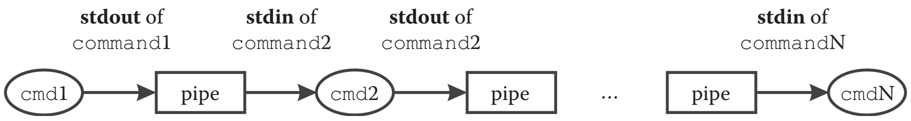


FIGURE 9.11 Semantics of a pipeline with *N* commands.

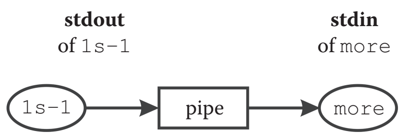


FIGURE 9.12 Semantics of the `ls -l | more` command.

```
[contents of temp]
$ rm temp
$
```

As you can see, not only do you need three commands to accomplish the same task, but the command sequence is also extremely slow because file read and write operations are involved. Recall that files are stored on a secondary storage device, usually a disk. On a typical contemporary computer system, disk operations are about one million times slower than main memory (random access memory [RAM]) operations. The actual performance gain in favor of pipes, however, is much smaller, owing to efficient caching of file blocks by the UNIX kernel and the use of semiconductor disks.

You can use the `sort` utility discussed in Chapter 7 to sort lines in a file. Suppose that you have a file called **student_records** that you want to sort and that the file may have some repeated lines that you want to appear only once in the sorted file. The `sort -u student _ records` command can accomplish this task. As we discussed in Chapter 6, the `uniq` command can also do the task if it is given the sorted version of **student_records** with repeated lines in it. One way to perform the task is to use the following commands. The `more` command is used to show the contents of **student_records**.

```
$ more student_records
John Doe     ECE   3.54
Pam Meyer    CS    3.61
Jim Davis    CS    2.71
John Doe     ECE   3.54
Jason Kim    ECE   3.97
Amy Nash     ECE   2.38
$ sort student_records > student_records_sorted
$ uniq student_records_sorted
Amy Nash     ECE   2.38
Jason Kim    ECE   3.97
Jim Davis    CS    2.71
John Doe     ECE   3.54
Pam Meyer    CS    3.61
$
```

The same task can be accomplished in one command line by using a pipe, as follows:

```
$ sort student_records | uniq
Amy Nash     ECE   2.38
```

```
Jason Kim   ECE   3.97
Jim Davis   CS    2.71
John Doe    ECE   3.54
Pam Meyer   CS    3.61
$
```

At times, you may need to connect several commands. The following command line demonstrates the use of multiple pipes, forming a *pipeline* of commands. In this command line, we have used the `grep` and `sort` filters.

```
$ who|sort|grep"john"|mail -s "John's Terminal" doe@coldmail.com
$
```

This command sorts the output of `who` and sends the lines containing the string "john" (if any exists) as an e-mail message to **doe@coldmail.com**, with the subject line "John's Terminal". In terms of input and output redirection, this command line is equivalent to the following command sequence.

```
$ who > temp1
$ sort < temp1 > temp2
$ grep "john" temp2 > temp3
$ mail -s "John's Terminal" doe@coldmail.com < temp3
$ rm temp1 temp2 temp3
$
```

The command with pipes does not use any disk files, but the preceding command sequence needs three temporary disk files and six disk I/O (read and write) operations. The number of I/O operations may be a lot higher, depending on the sizes of these files, the system load in terms of the number of users currently using the system, and the runtime behavior of other processes running on the system.

A pipe, therefore, is a UNIX feature that allows two UNIX commands (processes) to communicate with each other. Hence, a pipe provides an interprocess communication (IPC) mechanism in UNIX. More specifically, it can be used as a channel between two related processes on the same system to talk to each other. Typically, processes have a parent–child relationship (see Chapter 10) but processes with a common ancestor (parent, grandparent, etc.) can also communicate using a pipe. From a programmer's perspective, we discuss IPC using pipes in detail in Chapter 19. The communication between processes is one-way only. For example, in `ls | more`, the output of `ls` is read by `more` as input. Thus, the one-way communication is from `ls` to `more`. For a two-way communication between processes, at least two pipes must be used. This cannot be accomplished at the command shell level, but it can be done in C/C++ by using the `pipe( )` system call. We explore this topic in Chapter 19 also.

I/O redirection and pipes can be used in a single command, as follows:

```
$ grep "John" < Students | lpr -Pspr
$
```

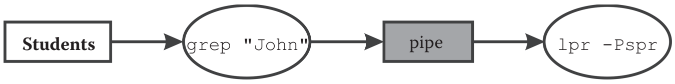


FIGURE 9.13 Semantics of the `grep "John" < Students | lpr -Pspr` command.

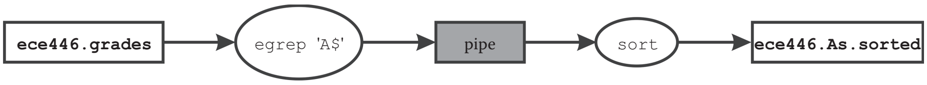


FIGURE 9.14 Semantics of the `egrep 'A$' < ee446.grades | sort > ee446.As.sorted` command.

Here, the `grep` command searches the **Students** file for lines that contain the string "John" and sends those lines to the `lpr` command to be printed on a printer named **spr**. Figure 9.13 illustrates the semantics of this command.

In the following command, `egrep` takes its input from **ee446.grades** and sends its output (lines ending with the character A) to the `sort` utility, which sorts these lines and stores them in the file called **ee446.As.sorted** in the current directory. The end result is that the names, scores, and grades of those students who have A grades in the ECE446 course are stored in the **ee446.As.sorted** file in the current directory. Figure 9.14 illustrates the semantics of this command.

```
$ egrep 'A$' < ee446.grades | sort > ee446.As.sorted
$
```

Suppose that, before running the `ssh server sort < datafile` command in Section 9.4, you want to be sure that **datafile** on your local system is consistent with the updated copy on the server, called **~/research/pvm/datafile.server**. You can copy **datafile.server** and compare it with your local copy, **datafile**. But, you will then have three copies, and if you are not careful you might remove the wrong copy. In this case, you can run the following command to see the differences between your local copy and the copy on the server without copying **datafile.server** on your (local) computer.

```
$ ssh server cat ~/research/pvm/datafile.server | diff datafile -
$
```

In this case, the `cat` command runs on the server side, and its output is fed as input to the `diff` command executed on the local machine. The output of the `diff` command also goes to the display screen on the local machine. Figure 9.15 illustrates the semantics of this command.

9.12 REDIRECTION AND PIPING COMBINED

You cannot use the redirection operators and pipes alone to redirect the **stdout** of a command to a file as well as connect it to **stdin** of another command in a single command line. However, you can use the `tee` utility to do so. You can use this utility to tell the command shell to send the **stdout** of a command to one or more files specified as arguments of `tee`,

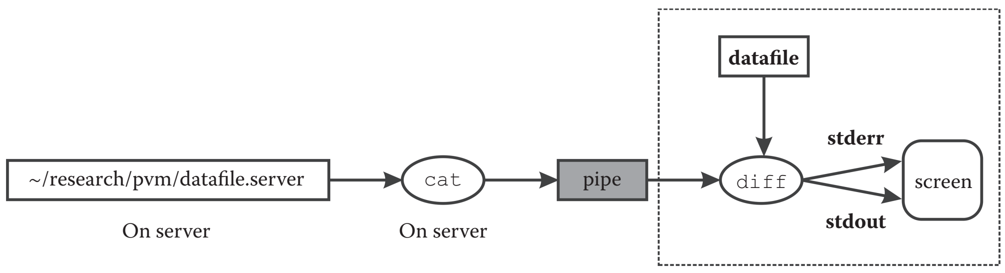


FIGURE 9.15 Semantics of the `ssh server cat ~/research/pvm/datafile.server | diff datafile -` command.

as well as to another command. The following is a brief description of how the `tee` utility is normally used.

SYNTAX

```
command1 | tee file1 file2 … fileN | command2
```

Purpose: Standard output of `command1` is connected to the **stdin** of `tee`, and `tee` sends its input to files **file1** through **fileN** as well as the **stdin** of `command2`

The semantics of this command syntax are that `command1` is executed and its output is stored in files **file1** through **fileN** as well as sent to `command2` as its input. An example use of the `tee` utility is given in the following command.

```
$ cat names students | grep "John Doe" | tee file1 file2 | wc -l
$
```

This command extracts the lines from the **names** and **students** files that contain the string “John Doe”, pipes these lines to the `tee` utility, which puts copies of these lines into **file1** and **file2** as well as sending them to `wc -l`, which sends its output to the display screen. Thus, the lines in the **names** and **students** files that contain the string “John Doe” are saved in **file1** and **file2**, and the line count of such lines is displayed on the monitor screen. Figure 9.16 illustrates the semantics of this command line. Such commands are useful in a shell script where different operations have to be performed on **file1** and **file2** later in the script.

9.13 OUTPUT AND ERROR REDIRECTION IN THE C SHELL

The operators for performing the input, output, and append operations (<, >, >>) work in the C shell as they do in other shells, as previously discussed. However, file descriptors cannot be used with these operators in the C shell. Also, error redirection works differently in the C shell than it does in other shells. In the C shell, the operator for output and error redirection is `>&`.

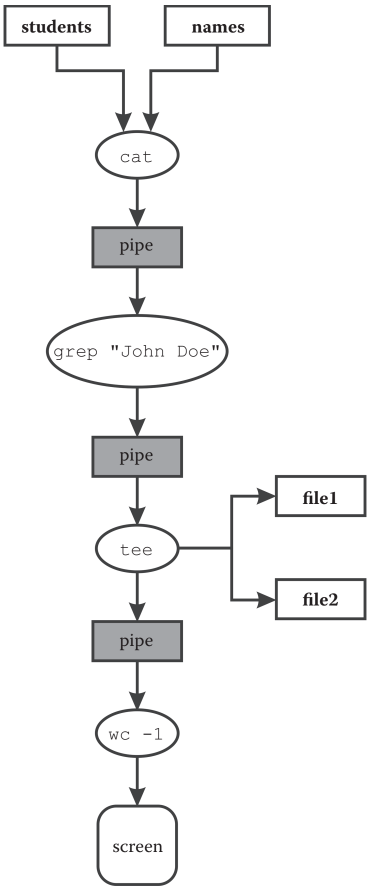


FIGURE 9.16 Semantics of the `cat names students | grep "John Doe" | tee file1 file2 | wc -l` command.

SYNTAX

`command >& file`

Purpose: Redirect **stdout** and **stderr** of `command` to **file**

For example, the following command redirects output and errors of the `ls -l foo` command to the **error.log** file. The standard input of the command is still attached to the keyboard. Note that we have used the `%` sign as the shell prompt, which is the default for the C shell.

```
% ls -l foo >& error.log
%
```

The C shell does not have an operator to redirect **stderr** alone. However, the **stdout** and **stderr** of a command can be attached to different files, if the command is executed in a subshell, by enclosing the command in parentheses. The following session illustrates this point.

```
% find ~ -name foo -print >& output.error.log
% (find ~ -name foo -print > foo.paths) >& error.log
%
```

The children of your current shell process, also known as subshells (see Chapter 10), execute all external shell commands. When the first command executes, the output and errors of the `find` command go to the **output.error.log** file. Because the subshell process is not created until the whole command line has been processed (interpreted), the **stdout** and **stderr** of the parent shell process are redirected to the **error.log** file because of the `>&` operator. Therefore, the subshell also has its **stdout** and **stderr** redirected to the **error.log** file.

In the second command line, the `find` command is executed under a subshell and inherits the standard files of the parent shell. When the `find` command in parentheses executes, it redirects the **stdout** of the command to the **foo.paths** file; the **stderr** of the command remains attached to **error.log**. Thus, the output of the `find` command goes to the **foo.paths** file, and the errors generated by the command go to the **error.log** file. Figure 9.17 illustrates the semantics of the second `find` command.

You can use the `>>&` operator to redirect and append **stdout** and **stderr** to a file. For example, `ls -l foo >>& output.error.log` redirects the **stdout** and **stderr** of the `ls` command and appends them to the **error.log** file.

The C shell also allows the **stdout** and **stderr** of a command to be attached to the **stdin** of another command with the `|&` operator. The following is a brief description of this operator.

SYNTAX

`Command1 |& command2`

Purpose: Redirect **stdout** and **stderr** of `command1` to `command2`, that is, pipe **stdout** and **stderr** of `command1` to `command2`

In the following command, the **stdout** and **stderr** of the `cat` command are attached to the **stdin** of the `grep` command. Thus, the output of the `cat` command, or any error produced by it (e.g., owing to the lack of read permission for **file1** or **file2**), is fed as input to the `grep` command.

```
% cat file1 file2 |& grep "John Doe"
%
```

The I/O redirection and piping operators (`|` and `|&`) can be used in a single command, as shown in the following session. This command is an extension of the previous command,

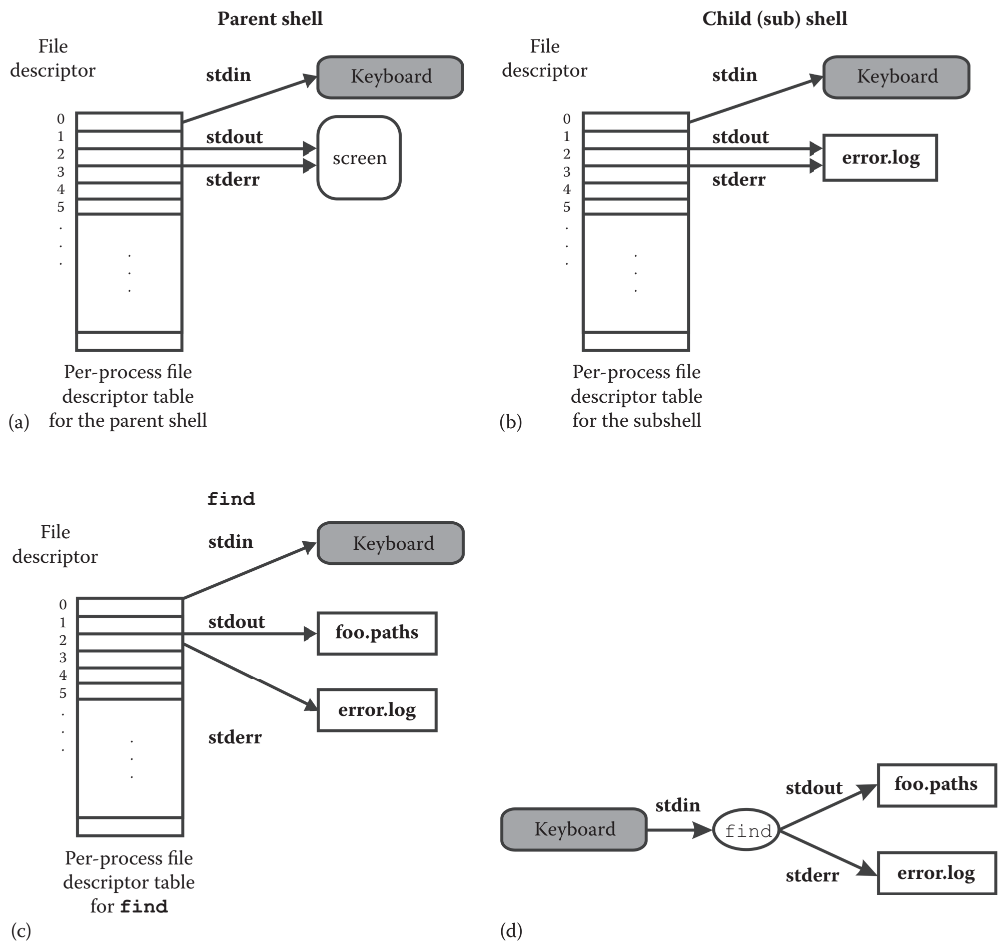


FIGURE 9.17 Step-by-step semantics of the `(find ~ -name foo -print > foo.paths) >& error.log` command.

in which the **stdout** of the `grep` command is attached to the **stdin** of the `sort` command. Furthermore, the **stdout** and **stderr** of the `sort` command are attached to the **stdin** of the `wc -l` command. Thus, if the command line completes successfully, it display the number of lines in **file1** and **file2** that contain the string "John Doe".

```
% cat file1 file2 |& grep "John Doe" | sort |& wc -l
%
```

In the following in-chapter exercises, you will practice the use of UNIX pipes, the `tee` command, and the error redirection feature of the C shell.

EXERCISE 9.6

Write a shell command that sorts a file **students.records** and stores the lines containing "Davis" in a file called **Davis.record**. Use piping and I/O redirection.

EXERCISE 9.7

Write a command to copy a file **Scores** to **Scores.bak** and send a sorted version of **Scores** to **professor@university.edu** via the `mail` command.

EXERCISE 9.8

Write a C shell command for copying a file **Phones** in your home directory to a file called **Phones.bak** (also in your home directory) by using the `cat` command and the `>&` operator.

The C shell has a special built-in variable that allows you to protect your files from being overwritten with output redirection. This variable is called `noclobber` and, when set, prevents the overwriting of existing files with output redirection. You can set the variable by using the `set` command and unset it by using the `unset` command. Or, you can place the `set noclobber` command in your **~/.cshrc file** (or some other startup file).

```
% set noclobber
[your interactive session]
...
% unset noclobber
%
```

If the `noclobber` variable is set, the command `cat file1 > file2` generates an error message if **file2** already exists. If **file2** does not exist, it is created and the data from **file1** is copied into it. The command `cat file1 >> file2` works if **file2** exists and `noclobber` is set, but an error message is generated if **file2** does not exist. You can use the `>!`, `>>!`, and `>>&!` operators to override the effect of the `noclobber` variable if it is set. Therefore, even if the `noclobber` variable is set and **file2** exists, the command `cat file1 >! file2` copies data from **file1** to **file2**. For the `cat file1 >>! file2` command, if the `noclob-ber` variable is set and **file2** does not exist, **file2** is created and the data from **file1** is copied into it. The `>>&!` operator works in a manner similar to the `>>!` operator.

9.14 RECAP OF I/O AND ERROR REDIRECTION

Table 9.1 summarizes the input, output, and error redirection operators in the Bourne, Korn, and C shells. We did not discuss some of these operators in this chapter; we discuss them in detail in Chapters 12 through 15 under shell programming. We included these operators in this table because it seems to be the most appropriate place to show all of them together. Note that all of the operators that work for the Bourne and Korn shells work for the Bash shell also.

9.15 FIFOS

FIFOs, also known as *named pipes*, can also be used for communication between two processes executing on a system. Whereas processes communicating with pipes must be related to each other through a common ancestor process that you execute, processes

TABLE 9.1 Redirection Operators and Their Meaning in the Bourne, Korn, and C Shells

Operator	Bourne Shell	Korn Shell	C Shell
`< file`	Input redirection	Input redirection	Input redirection
`> file`	Output redirection	Output redirection	Output redirection
`>> file`	Append standard standard output	Append standard output	Append output
`0< file`	Input redirection	Input redirection	
`1> file`	Output redirection	Output redirection	
`2> file`	Error redirection	Error redirection	
`1>> file`	Append standard output to **file**	Append standard output to **file**	
`2>> file`	Append standard error to **file**	Append standard error to **file**	
`<&m`	Attach standard input to file descriptor `m`	Attach standard input to file descriptor `m`	
`>&m`	Attach standard output to file with descriptor `m`	Attach standard output to file with descriptor `m`	
`m>&n`	Attach file descriptor `m` to file descriptor `n`	Attach file descriptor `m` to file descriptor `n`	
`<&-`	Close standard input	Close standard input	
`>&-`	Close standard output	Close standard output	
`m<&- or m>&-`	Close file descriptor `m`	Close file descriptor `m`	
`>& file`			Output and error redirection
`>| file`		Ignore `noclobber` and assign standard output to **file**	
`>>| file`		Ignore `noclobber` and append standard output to **file**	
`>! file`			Ignore `noclobber` and assign standard output to **file**
`>>! file`			Ignore `noclobber` and append standard output to **file**; if **file** does not exist, create it
`>>&! file`			Ignore `noclobber` and append standard output and standard error to **file**
`cmd1 | cmd2`	Connect standard output of command `cmd1` to standard input of command `cmd2`	Connect standard output of command `cmd1` to standard input of command `cmd2`	Connect standard output of command `cmd1` to standard input of command `cmd2`

(*Continued*)

TABLE 9.1 (Continued) Redirection Operators and Their Meaning in the Bourne, Korn, and C Shells

Operator	Bourne Shell	Korn Shell	C Shell
`cmd1 |&cmd2`			Connect standard output and standard error of command `cmd1` to standard input of command `cmd2`
`(cmd>/dev/tty)>&file`			Redirect standard error of the `cmd` command to **file**.
`|&`			Allow **stdout** and **stderr** of a command to be attached to **stdin** of another command

communicating with FIFOs do not have to have this kind of relationship—they can be independently executed programs on one system. For the command line use of pipes and FIFOs, this means that pipes can be used only for communication between commands connected via a pipeline and FIFOs can be used for communication between separately run commands. Another difference between pipes and FIFOs is that whereas a pipe is a main memory buffer maintained by the UNIX kernel and has no name, a FIFO is created on disk and has a name like a filename. This means that, like files, FIFOs have to be created and opened before they can be used for communication between processes. Thus, accessing a FIFO requires an access to the secondary storage device where it resides. Pipes are *process persistent*, which means that they exist as long as the process that creates them exists. FIFOs on most UNIX systems are *filesystem persistent*, meaning that they exist on the system until they are explicitly removed/deleted or the relevant file system is unmounted (i.e., removed from the system).

You can use the `mknod()`system call or the `mkfifo()`library call to create a FIFO in a program and the `mkfifo` command to create a file in a shell session. We discuss the command line use of FIFOs in this section. We discuss the use of FIFOs, pipes, and the related system calls and library calls in the chapters on system programming (Chapters 18 through 21). Here is a brief description of the `mkfifo` command.

SYNTAX

```
mkfifo [option] file-list
```

Purpose: Create FIFOs with pathnames given in **file-list**
Output: FIFOs for the pathnames given in **file-list** are created in the relevant directories
Commonly used options/features:

-m mode Set access permissions for newly created FIFOs to "mode"; the access permissions are specified in "mode" as they are with the `chmod` command, such as 666, meaning read and write permissions for everyone and execute permission for nobody

In the following session, we use the first command to create a FIFO, called **myfifo1**, with default permissions based on the current value of `umask` (see discussion in Chapter 5). We use the second command to create a FIFO, called **myfifo2**, with read and write permissions for the owner and no permissions for all other users. We use the `ls -al myfifo1 myfifo2` command to display the access permissions of the two FIFOs. Note that the first character in the long listing of the two FIFOs is "p", indicating that **myfifo1** and **myfifo2** are of the FIFO (named pipe) type.

```
$ mkfifo myfifo1
$ mkfifo -m 600 myfifo2
$ ls -l myfifo3 myfifo4
prw-r--r-- 1 sarwar faculty 0 Aug 9 08:30 myfifo3
prw------- 1 sarwar faculty 0 Aug 9 08:31 myfifo4
$
```

The general method by which two commands, `cmd1` and `cmd2`, can communicate with a FIFO, called **myfifo1**, is shown in the following command sequence.

```
cmd1 < myfifo1 &
cmd2 infile | tee myfifo1 | cmd3
```

Note that we run the first command in the background (see Chapter 10 for background processes) so that we could run `cmd2`. When we execute `cmd1`, it blocks, because **myfifo1** is empty. Note that `cmd1` blocks and returns immediately if **myfifo1** is a file. When the output of `cmd2` is sent to **myfifo1** and `cmd3` via the `tee` utility, `cmd1` unblocks and starts processing data in **myfifo1**. Outputs of commands `cmd1` and `cmd3` are sent to standard output (i.e., the display screen). Figure 9.18 shows these semantics with the help of a diagram.

In the following session, the command sequence displays the status of all the processes running on the system, the number of daemon processes, and the total number of processes running on the system. The two `cat` commands block until something is written into **myfifo1** and **myfifo2**. The `ps -a` command sends the status of all the processes running on the system to the `tee` command, which redirects this data to the two FIFOs as well as sending it to the `grep` command. The `grep` command extracts all the daemon processes from its input (i.e., the output of the `ps -a` command) and sends them to the `wc -l` command through a pipe. Thus, the first `cat` command displays the status of all the processes running

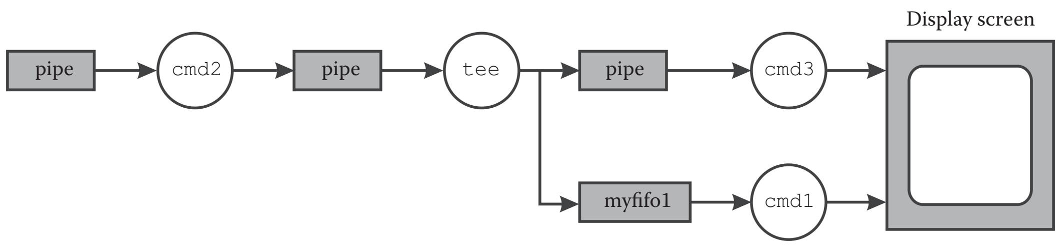


FIGURE 9.18 Semantics of execution of the command sequence `cmd1 < myfifo1 &` followed by `cmd2 infile | tee myfifo1 | cmd3`.

on the system. The output of the second `cat` command is the number of processes running on the system and that of the third command (`ps –a`) is the number of daemons running on the system. As shown by the last two lines of the output, at the time of running this command sequence, the system was running 20 processes, of which two were daemons.

```
$ cat myfifo1 &
$ cat myfifo2 | wc -l &
$ ps -a | tee myfifo1 myfifo2 | grep 'd$' | wc -l
  PID TT  STAT   TIME COMMAND
 1533 v0- I   0:00.01 /bin/sh /usr/local/sbin/PCDMd
 1548 v0- IN  3:43.82 /bin/sh /usr/local/sbin/pbid
 1790 v0  Is+ 0:00.00 /usr/libexec/getty Pc ttyv0
 1791 v1  Is+ 0:00.00 /usr/libexec/getty Pc ttyv1
 ...
 1797 v7  Is+ 0:00.00 /usr/libexec/getty Pc ttyv7
56478  1  Is  0:00.24 -csh (csh)
57216  1  S   0:00.12 /bin/sh
58002  1  S   0:00.01 cat myfifo1
58045  1  S   0:00.01 cat myfifo2
58046  1  I   0:00.01 wc -l
58089  1  R+  0:00.02 ps -a
58090  1  S+  0:00.01 tee myfifo1 myfifo2
58091  1  S+  0:00.02 grep d$
58092  1  S+  0:00.01 wc -l
       2
$      20
```

The sequence of the output in this session is dependent on the scheduling of the three commands; the output shown above is what was produced by our system and is the most likely output. An interesting exercise would be to come up with a sequence of commands to ensure that the output is always produced in the same order as seen here.

When you no longer need to use a FIFO, you can remove it just like you remove an ordinary file. This means that you can use the `unlink()`system call (from within a process) or the `rm` command at the command line for removing a FIFO from your file system hierarchy. In the following session, we use the `rm` command for removing the **myfifo1** and **myfifo2** FIFOs. The output of the `ls` command before and after the `rm` command shows that the two FIFOs have in fact been removed.

```
$ ls
myfifo1 myfifo2
$ rm myfifo1 myfifo2
$ ls
$
```

The following in-chapter exercises are designed to give you practice using the `mkfifo` command and help you to understand its semantics with a hands-on session.

EXERCISE 9.9

Create three FIFOs, called **fifo1**, **fifo2**, and **fifo3**, with a single command. Write down the command that you used to perform the given task.

EXERCISE 9.10

Create a FIFO, called **fifo4**, with its access privileges set to read and write for owner and group, and no privileges for others. Show the command that you used to accomplish the given task.

EXERCISE 9.11

Try the shell session given in this section on your system. Does your system produce output in the same order as shown in our session? If not, show the output produced on your system.

SUMMARY

UNIX automatically opens three files for every command for it to read input from and send its output and error messages to. These files are called standard input (**stdin**), standard output (**stdout**), and standard error (**stderr**). By default, these files are attached to the terminal on which the command is executed. Thus, the shell makes the command input come from the keyboard and its output and error messages to go to the monitor screen. These default files can be changed to other files by using redirection operators: < for input redirection and > for output and error redirection.

The three standard files can be referred to by using the digits 0 (**stdin**), 1 (**stdout**), and 2 (**stderr**), called the file descriptors for the three standard files. All open files in UNIX are referred to by similar integers that are used by the kernel to perform operations on these files. In the Bourne, Korn, Bash, and POSIX shells, the greater-than symbol (>) is used in conjunction with descriptors 1 and 2 to redirect standard output and standard error, respectively.

The standard output of a command can be connected to the standard input of another command via a UNIX pipe (|). Pipes are created in the main memory and are used to take the output of a command and give it to another command without creating a disk file, effectively making the two commands talk to each other. For this reason, a pipe is called an interprocess communication (IPC) channel, which allows related commands on the same machine to communicate with each other at the shell and application levels. The processes communicating through pipes must be related through a common ancestor; the relationship is usually parent–child or sibling.

The I/O and error redirection features and pipes can be used together to implement powerful command lines. However, redirection operators and pipes alone cannot be used to redirect the standard output of a command to a file as well as connecting it to standard input of another command. The `tee` utility can be used to accomplish this task, sending standard output of a command to one or more files as well as to another command. The commands and `tee` are connected through pipes.

The C shell does not support I/O and error redirection with file descriptors. Also, redirecting standard output and standard error of a command to different files is specified differently in the C shell than it is in the other shells.

FIFOs, also known as named pipes, allow related or unrelated processes on a system to communicate. Unlike a pipe, which is an in-memory buffer, a FIFO is a file created on a secondary storage device. For command line use, you can create a FIFO with the `mkfifo` command. The `mknod()` system call or `mkfifo()` library call may be used to create FIFOs within processes. When you no longer need a FIFO, you can remove it with the `unlink()` system call from within a running program (process) or the `rm` command at the command line. We discuss the UNIX system calls and library calls related to pipes and FIFOs in Chapters 18 through 21 on UNIX system programming.

QUESTIONS AND PROBLEMS

1. What are standard files? Name them and state their purpose.
2. Briefly describe input, output, and error redirection. Write two commands of each to show simple and combined use of the redirection operators.
3. What are file descriptors in UNIX? What are the file descriptors of standard files? How can the I/O and error redirection operators be combined with the file descriptors of standard files to perform redirection in the Bourne, Korn, Bash, and POSIX shells?
4. Sort a file **data1** and put the sorted file in a file called **data1.sorted**. Give the command that uses both input and output redirection.
5. Give the command to accomplish the task in Problem 4 by using a pipe and output redirection.
6. Give a set of commands equivalent to the command `ls -l | grep "sarwar" > output.p3` that use I/O redirection operators only. How does the performance of the given command compare with your command sequence? Explain.
7. What is the purpose of the `tee` command? Give a command equivalent to the command in Problem 6 that uses the `tee` command.
8. Write UNIX shell commands to carry out the following tasks.
   a. Count the number of characters, words, and lines in a file called **data1** and display the output on the display screen.
   b. Count the number of characters, words, and lines in the output of the `ls -l` command and display the output on the display screen.
   c. Do the same as in part (b), but redirect the output to a file called **data1.stats**.

9. Give the command for searching a file **datafile** for the string "Internet", sending the output of the command to a file called **Internet.freq** and any error message to a file **error.log**. Draw a diagram for the command, similar to the ones shown in the chapter, to illustrate its semantics.
10. Give a command for accomplishing the task in Problem 9, except that both the output of the command and any error message go to a file called **datafile**.
11. Give a command to search for lines in **/etc/passwd** that contain the string "sarwar".
12. Store the output of the command in a file called **passwd.sarwar** in your current directory. If the command fails, the error message must also go to the same file.
13. What is the UNIX pipe? How is pipe different from output redirection? Give an example to illustrate your answer.
14. What do the following commands do under the Bourne shell?
    a. `cat 1> letter 2> save 0< memo`
    b. `cat 2> save 0< memo 1> letter`
    c. `cat 1> letter 0< memo 2>&1`
    d. `cat 0< memo | sort 1> letter 2> /dev/null`
    e. `cat 2> save 0< memo | sort 1> letter 2> /dev/null`
15. Consider the following commands under the Bourne shell.
    a. `cat memo letter 2> communication 1>&2`
    b. `cat memo letter 1>&2 2> communication`

    Where do output and error messages of the `cat` command go in each case if
    a. both files (memo and letter) exist in the present working directory, and
    b. one of the two files does not exist in the present working directory?
16. Send an e-mail message to **doe@domain.com**, using the mail command. Assume that the message is in a file called **greetings**. Give one command that uses input redirection and one that uses a pipe. Any error message should be appended to a file **mail.error**.
17. What happens when the following commands are executed on your UNIX system? Why do these commands produce the results that they do?
    a. `cat letter >> letter`
    b. `cat letter > letter`

18. By using output redirection, send a greeting message "Hello, World!" to a friend's terminal.
19. Give a command for displaying the number of users currently logged on to a system.
20. Give a command for displaying the login name of the user who was the first to log on to a system.
21. What is the difference between the following commands?
    a. `grep "John Doe" Students > /dev/null 2>&1`
    b. `grep "John Doe" Students 2>&1 > dev/null`
22. Give a command for displaying the contents of (the files' names in) the current directory, with three files per line.
23. Give a command that reads its input from a file called **Phones**, removes unnecessary spaces from the file, sorts the file, and removes duplicate lines from it.
24. Repeat Problem 23 for a version of the file that has unnecessary spaces removed from the file but still has duplicate lines in it.
25. What do the following commands do?
    a. `uptime | cat - who.log >> system.log`
    b. `zcat secret_memo.Z | head -5`
26. Give a command that performs the task of the following command but with the `diff` command running on the machine called **server**: `ssh server cat ~/research/pvm/datafile.server | diff datafile -`
27. Give a command for displaying the lines in a file called **employees** that are not repeated. What is the command for displaying repeated lines only?
28. Give a command that displays a long list for the most recently created directory.
29. Create a FIFO, called **myfifo1**. What are the default access privileges set for it? Create a FIFO, **myfifo2**, with read and write access privileges for everyone. Show your commands and their output for performing these tasks.
30. Give a set of commands for producing the output of the session given in Section 9.15. Your command sequence should ensure that the order of output is always the following: the status of all the processes running on the system, the number of daemons running on the system, and the total number of processes running on the system.
31. The number of files a process can open simultaneously on a UNIX system is dependent on the size of the per-process file descriptor table on the system. Use a shell command with the features discussed in this chapter to display the value of the

_POSIX_OPEN_MAX variable in the **/usr/include/limits.h** file. Show your command and its output and explain your answer.

32. How much data can a pipe store on your system? Use a shell command to obtain your answer. What command did you use? Show the command and its output. *Hint*: Look for a symbolic constant defined in the **/usr/include/limits.h** file.
33. What is the output of the following command? Give reasons for your answer.

```
cat file1 file2 |& grep "John Doe" | sort |& wc -l
```

34. What is the purpose of the following command?

```
cat /usr/include/limits.h | nl | grep_POSIX_OPEN_MAX
```

35. What do the following commands do?
    a. `mail mike@somewhere.org < to_do`
    b. `cat Phones | sort | uniq | pr | lpr`

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

CHAPTER 10

Processes

Objectives

- To describe the concept of a process, and execution of multiple processes on a computer system with a single CPU
- To explain how a shell executes commands
- To discuss static and dynamic display of process attributes
- To discuss the main memory image of a UNIX process
- To describe briefly the concept of CPU scheduling and scheduling classes in UNIX
- To explain the concept of foreground and background processes, including a description of a daemon and its uses
- To describe sequential and parallel execution of commands
- To discuss process and job control in UNIX: foreground and background processes, sequential and parallel processes, suspending processes, moving foreground processes into the background and vice versa, and terminating processes
- To describe the UNIX process hierarchy
- To cover the commands and primitives

```
<Ctrl+C>, <Ctrl+D>, <Ctrl+Z>, <Ctrl+\>, ;, &, ( ), bg, fg,
jobs, kill, nice, nohup, pagezise, ps, ptree, size, sleep, top
```

10.1 INTRODUCTION

As we have mentioned before, a process is a program in execution. The program may be assembly language code, an executable code generated after compiling a source program written in a high-level language such as C++, or an interpreted code written in LISP, JavaScript, Perl, Interpreted C (CINT), or a UNIX shell. The UNIX system creates a process every time you run an external command, and the process is removed from the system when the command finishes its execution. We use the terms *program* and *command* interchangeably.

Process creation and termination are the only mechanisms used by the UNIX system to execute external commands. In a typical time-sharing system such as UNIX, which allows multiple users to use a computer system and run multiple processes simultaneously, hundreds to thousands of processes are created and terminated every day. Remember that the CPU in the computer executes processes and that a typical system has only one CPU. The question is, how does a system with a single CPU execute multiple processes simultaneously? Even for systems with multiple CPUs or multiple cores in a CPU, the number of processes is greater than the number of CPUs or cores. How does a system with the number of processes larger than the number of CPUs or CPU cores execute these processes simultaneously? A detailed discussion of this topic is beyond the scope of this textbook, but we briefly address it in Section 10.2 and later in Section 10.5.1. Later in the chapter, we discuss viewing the static and dynamic state of processes, foreground and background processes, daemons, jobs, process and job attributes, and process and job control. We use the terms *time sharing* and *multitasking* synonymously.

10.2 CPU SCHEDULING: RUNNING MULTIPLE PROCESSES SIMULTANEOUSLY

On a typical computer system that contains a single CPU and runs a time-sharing operating system, multiple processes are simultaneously executed by quickly switching the CPU from one process to the next. That is, one process is executed for a short period of time, and then the CPU is taken away from it and given to another process. The new process executes for a short period of time and then the CPU is given to the next process. This procedure continues until the first process in the sequence gets to use the CPU again. The time a process is "in" the CPU before it is switched "out" of the CPU is called a *quantum* or *time slice*. The quantum is usually very short: one second or less for a typical UNIX system. On Solaris, the quantum of a process is dependent on the priority of a process. For time-sharing user processes, the quantum is 40 milliseconds. For higher priority processes, the quantum value is larger. For example, the quantum is 200 milliseconds for priority 0 kernel threads and 160 milliseconds for priority 10 kernel threads. In FreeBSD, the quantum value is 0.1 seconds. When the CPU is free/idle (i.e., not used by any process) or when the current process has finished its quantum, the kernel uses an algorithm to decide which process gets to use the CPU next. The technique used to choose the process that gets to use the CPU is called *CPU scheduling*. The kernel code that performs this task is known as the *short-term CPU scheduler*, or *CPU scheduler*.

The process of taking the CPU away from the currently executing process and giving it to the newly scheduled process is called *context switching*. This task is performed by another part of the kernel, known as the *dispatcher*. In systems with multiple CPUs or CPUs with multiple cores such as those by Intel, AMD, and other companies, if the number of processes on the system is more than the number of CPUs in the system (or the number of cores for a single CPU system), CPU scheduling and context switching still happen. Thus, on a system with multiple users running multiple processes, the scheduler and dispatcher work in tandem to make you feel as if you are the only one using the system. Although a focused discussion on CPU-scheduling algorithms is beyond the scope of this book, we give a brief and simplistic view of how UNIX SV, FreeBSD, and Solaris schedulers work.

In a time-sharing system, a priority value is assigned to every process, and the process that has the highest priority gets to use the CPU next. Several methods can be used to assign a priority value to a process. One simple method is based on the time that it enters the system. In this scheme, typically the process that enters the system first is assigned the highest priority and gets to use the CPU next; the result is called a *first-come, first-serve* (FCFS) scheduling algorithm. Another scheme is to assign a priority value based on the amount of time a process has used the CPU. Thus, a newly arriving process, or a process that spends most of its time performing input and/or output (I/O) operations, gets the highest priority. Processes that spend most of their time performing I/O are known as *I/O-bound processes.* An example of an I/O-bound process is a text editor such as vim. In the *round robin* (RR) scheduling algorithm, the CPU is given to each process in the queue of processes for one quantum, one after the other. This algorithm is a natural choice for time-sharing systems, wherein all users like to see progress by their processes. If you are interested in other CPU scheduling algorithms, we encourage you to read a book on operating system principles and concepts. The operating system code that implements the CPU scheduling algorithm is known as the *processor scheduler.* The scheduler for most operating systems, including UNIX, is in the kernel.

The UNIX System V scheduling algorithm is a blend of all of the algorithms mentioned and more. It uses a simple formula to assign a priority value to every process in the system that is ready to run. The priority value for every process in the system is recalculated every second. When it is time for scheduling, the CPU is given to the process with the *smallest* priority number. If multiple processes have the same priority number, the decision is made on the FCFS basis. The formula used to compute the priority value is

$$\text{priority value} = \text{threshold priority} + \text{nice value} + \left(\text{recent CPU usage}/2\right),$$

where *threshold priority* is an integer usually having a value of 40 or 60, *nice value* is a positive integer with a default value of 10 but can be a value from –20 to 19 (20 on some UNIX systems), and *CPU usage* is the number of clock ticks (1/60 or 1/50 of a second on older systems, where 60 or 50 is the frequency of the power line in Hz) for which the process has used the CPU. On modern UNIX systems, a clock tick is much smaller and is not calculated based on the power-line frequency. The clock *interrupt service routine* (ISR) updates the CPU usage for every process every clock tick, which increases the tick count for the process currently using the CPU. The clock ISR divides the tick count of every process by two before it recalculates the process priorities by using the formula shown. This division by two is known as applying the *decay function* because it decreases the impact of previous CPU usage exponentially. Thus, recent CPU usage by a process has more impact on the priority of a process and historical CPU usage has a diminishing effect. The CPU usage value therefore increases for the process using the CPU and decreases for all other processes.

You can assign a higher nice value to your processes by using the `nice` or `renice` command, but the nice value cannot be set to a negative number by a nonsuperuser. A higher nice value means a higher priority value and, hence, a lower priority. So, when you increase the nice value of your process, you are being nice to other user processes. The formula clearly indicates that the higher the recent CPU usage of a process, the higher its priority value and the lower its priority. Thus, UNIX favors processes that have used less CPU time in the recent past. A text editor such as vim gets higher priority than a process that computes the value of pi (π) because vim spends most of the time waiting for I/O—that is, reading keyboard input, reading/writing to disk, and displaying file data or keyboard input on the screen. On the other hand, the process that computes π spends most of its time doing calculations—that is, using the CPU. Recalculating priority values of all the processes every second causes process priorities to change dynamically (up and down). In Section 10.5, we further explore the UNIX scheduling concept, particularly with respect to PC-BSD and Solaris.

10.3 UNIX PROCESS STATES

A UNIX process can be in one of many states, moving from one state to another, eventually finishing its execution, normally or abnormally, and getting out of the system. A process terminates normally when it finishes its work and exits the system gracefully. A process terminates abnormally when it exits the system because of an *exception* (error condition) or *intervention* by its owner or the superuser. The owner of the process can intervene by using a command or a particular keystroke to terminate the process. We discuss these commands and keystrokes later in the chapter. The primary states that a process can be in are shown in the state diagram in Figure 10.1.

The *waiting state* encompasses several states; we use the term here to keep the diagram simple. Some of the states belonging to the waiting state are listed under the oval representing the state. Table 10.1 gives a brief description of these UNIX process states. In the interest of brevity, and in keeping with the scope of this book, the other states that a UNIX process can be in are not included in this discussion.

10.4 EXECUTION OF SHELL COMMANDS

A shell command can be internal (built in) or external. An *internal/built-in command* is one whose code is part of the shell process. Some of the commonly used internal commands are `.` (dot command), `bg`, `cd`, `continue`, `echo`, `exec`, `exit`, `export`, `fg`, `jobs`, `pwd`, `read`, `readonly`, `return`, `set`, `shift`, `test`, `times`, `trap`, `umask`, `unset`, and `wait`. An *external command* is one whose code is in a file; contents of the file can be binary code or a shell script. Some of the commonly used external commands are `grep`, `more`, `cat`, `mkdir`, `rmdir`, `ls`, `sort`, `ftp`, `telnet`, `lp`, and `ps`. A shell creates a new process to execute a command. While the command process executes, the shell waits for it to finish. In this section, we describe how a shell (or any process) creates another

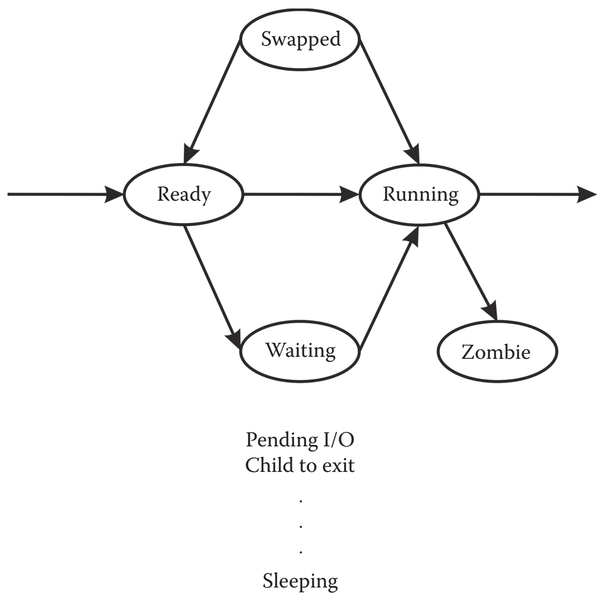


FIGURE 10.1 UNIX process state diagram.

TABLE 10.1 A Brief Description of the UNIX Process States

State	Description
Ready	The process is ready to run but does not have the CPU. Based on the scheduling algorithm, the scheduler decided to give the CPU to another process. Several processes can be in this state, but on a machine with a single CPU, only one can be executing/running (i.e., using the CPU).
Running	The process is actually running (i.e., using the CPU).
Waiting	The process is waiting for an event. Possible events are an I/O operation to complete (e.g., disk/terminal read or write), a child process to complete (the parent is waiting for one or more of its children to exit), or the process itself is waiting to be reawakened having been put to sleep.
Swapped	The process is ready to run, but it has been temporarily put on the disk (on the swap space); perhaps it needs more memory and there is not enough available at this time.
Zombie	A dying process is said to be in a zombie state. Usually, when the parent of a process terminates before it executes the exit call, it becomes a zombie process. The process finishes and finds that the parent is not waiting. The zombie processes are finished for all practical purposes and do not reside in the memory, but they still have some kernel resources allocated to them and cannot be taken out of the system. All zombies and their live children are eventually adopted by the granddaddy, the `init` process, which removes them from the system.

process and executes external commands. You can use the `type` command to determine if your command is built in or external, as shown in the following session. Under Bash, you can use the `–a` option to display all the locations of a command, as follows. You can see that the `bg` command is built in but also has an external version. However, by default, the built-in version is executed. The Bourne shell may be invoked through two executables available at two different places in your file system structure.

```
$ type bg
bg is a shell builtin
$ type -a bg
bg is a shell builtin
bg is /usr/bin/bg
$ type -a sh
sh is /usr/bin/sh
sh is /usr/sbin/sh
$
```

A UNIX process can create another process by using the `fork()` system call, which creates an exact main memory copy of the original process (i.e., the process that calls `fork()`). Both processes continue execution, starting with the statement that follows the fork. The forking process is known as the *parent process*, and the created (forked) process is called the *child process*, as shown in Figure 10.2. Here, we show a Bourne shell that has created a child process (another Bourne shell). We discuss the use of `fork()` and other system calls needed for the creation of processes and *interprocess communication* (IPC) in Chapters 18–21.

For executing an external binary command, a mechanism is needed that allows the child process to become the command to be executed. The UNIX system call `exec()` can be used to do exactly that, allowing a process to overwrite itself with the executable code for another command. A shell uses the `fork()` and `exec()` calls in tandem to execute an external binary command. Figure 10.3 shows the sequence of events for the execution of an external command `sort`, whose code is in a binary file, **/usr/bin/sort**.

The execution of a shell script (a series of shell commands in a file; see Chapters 12–15) is slightly different from the execution of a binary command/file. In the case of a shell script, the current shell creates a child shell and lets the child shell execute commands in the script file, one by one. Each command in the script file is executed in the same way that commands from the keyboard are; that is, the child shell creates a child for every

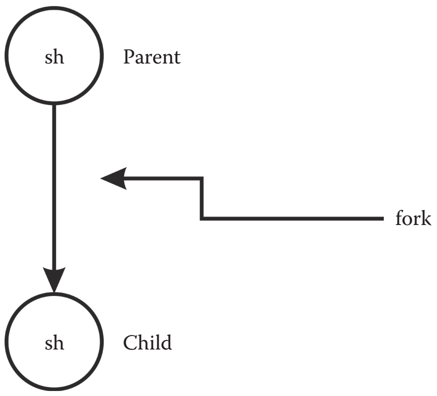


FIGURE 10.2 Process creation via the fork system call.

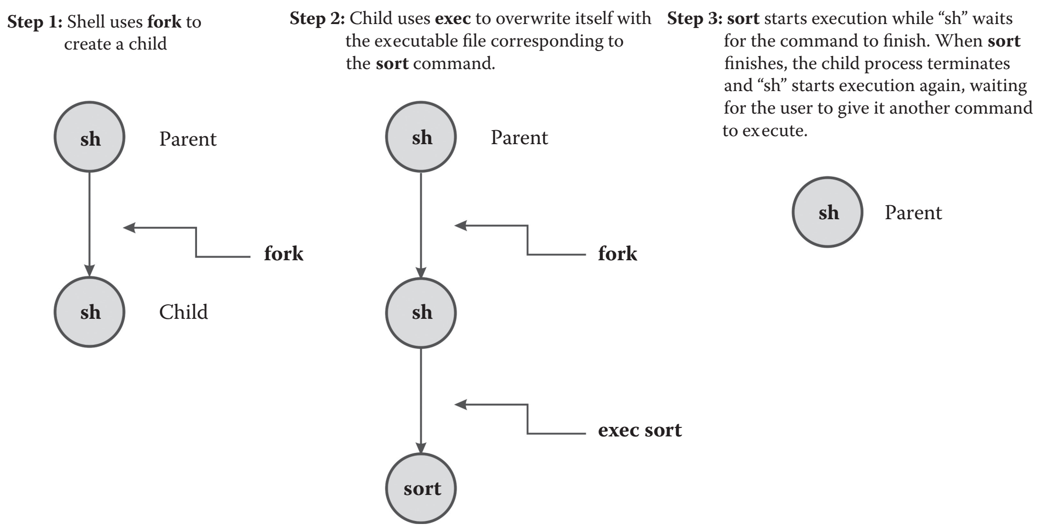

Step 1: Shell uses **fork** to create a child
sh
Parent
fork
sh
Child
Step 2: Child uses **exec** to overwrite itself with the executable file corresponding to the **sort** command.
sh
Parent
fork
sh
exec sort
sort
Step 3: **sort** starts execution while "sh" waits for the command to finish. When **sort** finishes, the child process terminates and "sh" starts execution again, waiting for the user to give it another command to execute.
sh
Parent

FIGURE 10.3 Steps for execution of a binary program `sort` by a UNIX shell.

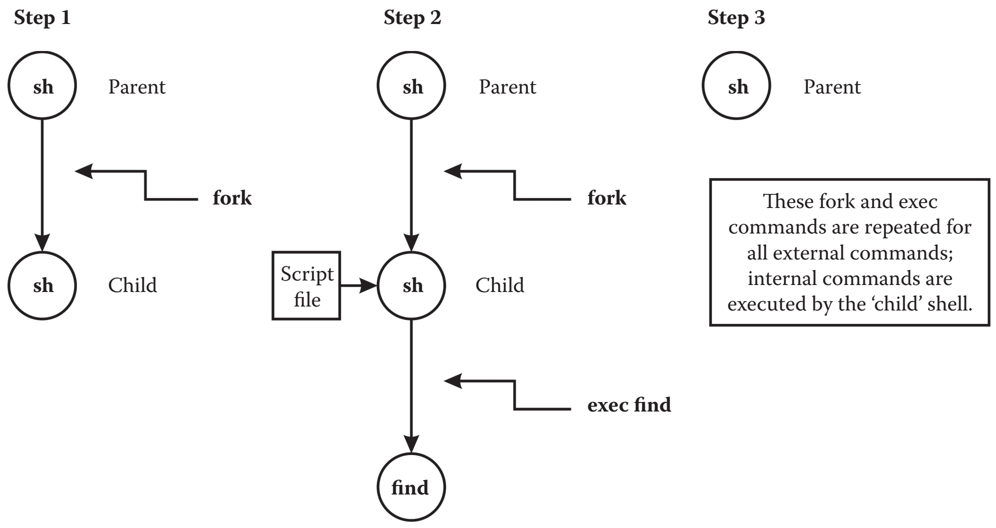

Step 1
sh
Parent
fork
sh
Child
Step 2
sh
Parent
fork
Script file
sh
Child
exec find
find
Step 3
sh
Parent
These fork and exec commands are repeated for all external commands; internal commands are executed by the 'child' shell.

FIGURE 10.4 Steps for execution of a shell script by a UNIX shell.

command that it executes. While the child shell is executing commands in the script file, the parent shell waits for the child to terminate. When the child shell hits the `eof` marker in the script file, it terminates. The only purpose of the child shell, like any other shell, is to execute commands, and `eof` means "no more commands." When the child shell terminates, the parent shell comes out of the waiting state and resumes execution. This sequence of events is shown in Figure 10.4, which also shows the execution of a `find` command in the script file.

Unless otherwise specified in the file containing the shell script, the child shell has the type of the parent shell. That is, if the parent is a Bourne shell, the child is also a Bourne shell. Thus, by default the shell script is executed by a “copy” of the parent shell. However, a shell script written for any shell (C, TC, Bourne, Bash, Korn, etc.) can be executed regardless of the type of the current shell. To do so, simply specify the type of the child shell under which the script should be executed in the first line of the file containing the shell script as `#!full-path- name-of-the-shell`. For example, the following line dictates that the child shell is C shell, so the script following this line is executed under the C shell.

```
#!/bin/csh
```

Also, you can execute commands in another shell by running that shell as a child of the current shell, executing commands under it, and terminating the shell. A child shell is also called a *subshell*. Recall that the commands to run various shells are `sh` for the Bourne shell, `csh` for the C shell, `tcsh` for the TC shell, `ksh` for the Korn shell, and `bash` for the Bourne Again shell. To start a new shell process, simply run the command corresponding to the shell you want to run.

In the following session, the current shell is the C shell and the Bourne shell runs as its child. The `echo` command is executed under the Bourne shell. Then a Bash shell is started, and the `echo` command is executed under it. The `ps` command shows the three shells running. Finally, both the Bash and Bourne shells are terminated when `<Ctrl+D>` is pressed in succession, and control goes back to the original shell, the C shell. The first `<Ctrl+D>` terminates the Bash shell, giving control back to the Bourne shell. You can also exit a shell running the `exit` command. Figure 10.5 illustrates all the steps involved, showing the parent–child relationship between processes.

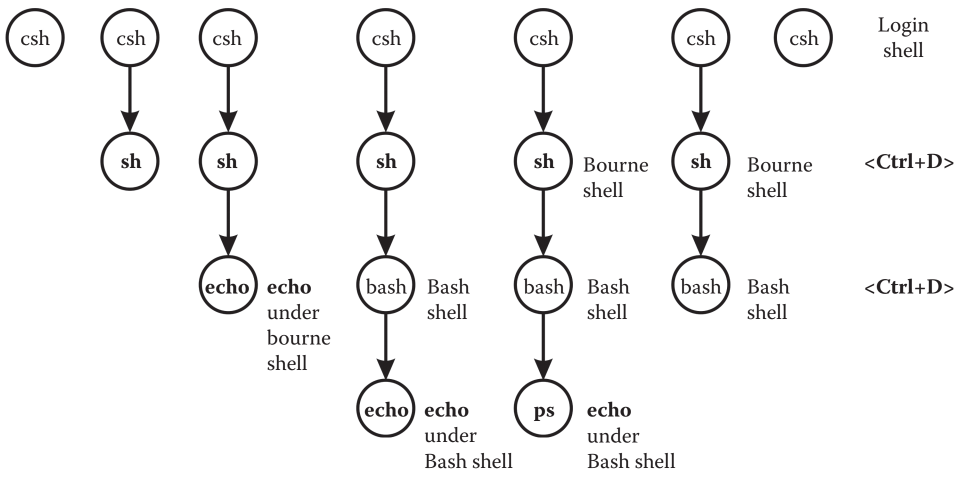


FIGURE 10.5 Execution of commands under the child shells (also called subshells).

```
% ps
  PID TT  STAT    TIME COMMAND
44387  5  Ss   0:00.97 -csh (csh)
45878  5  R+   0:00.01 ps
% /bin/sh
$ echo "This is Bourne shell."
This is Bourne shell.
$ bash
[sarwar@pcbsd-srv ~]$ echo "This is Bourne Again SHell."
This is Bourne Again SHell.
[sarwar@pcbsd-srv ~]$ ps
  PID TT  STAT    TIME COMMAND
44387  5  Is   0:00.97 -csh (csh)
45910  5  I    0:00.02 /bin/sh
45935  5  S    0:00.07 bash
46008  5  R+   0:00.01 ps
[sarwar@pcbsd-srv ~]$ <Ctrl+D>
$ <Ctrl+D>
%
```

10.5 PROCESS ATTRIBUTES

Every UNIX process has several attributes, including owner ID (called user ID [UID] in UNIX jargon), process ID (PID), PID of the parent process (PPID), process name, process state, command executed to start the process, priority of the process, process start time, percentage of the CPU time consumed by the process, percentage of the main memory consumed by the process, size of the process in virtual memory, size of the process currently in main memory, state of the process, the event a process is waiting for (in case it is not running), and the length of time the process has been running. From the user's and programmer's point of view, one of the most useful of these attributes is the PID, which is used as a parameter in several process control commands discussed later in this chapter.

UNIX provides several tools that allow you to monitor the attributes of the processes currently running on your system, change the states of your processes, and perform various operations on them, including stopping/restarting them, sending them specific messages, having them communicate with each other, and terminating them. In this chapter, we will discuss some of the commands and tools that allow us to monitor the attributes of processes statically and dynamically, as well as perform various operations on processes just stated.

10.5.1 Static Display of Process Attributes

The `ps` command can be used to view a snapshot of the attributes of processes currently in the system. PC-BSD and Solaris have their own versions of this command with several options. However, several of these commands are common. First, we discuss the PC-BSD version of the command in detail and then discuss the additional features of the Solaris version. The following is a brief description of the PC-BSD version of the `ps` command.

SYNTAX

`ps [options]`

Purpose: Report static (one shot) information about process status/attributes
Output: A header line and a snapshot of the attributes of processes running on the system
Commonly used options/features:

`-G`	Display information about processes running under user groups specified in the comma-separated list of group IDs; no space before or after the comma
`-H`	Display information about threads visible to the UNIX kernel
`-L`	Display the list of keywords that may be used with the –O or –o option
`-O`	Display information about the comma- or space-separated keywords after the PID field in the default output. You can assign the header of your choice for a keyword by putting an = after the keyword, followed by the header value.
`-U`	Display information about the processes for the users specified in the comma-separated list of usernames; no spaces before or after the commas
`-a`	Display information about your and other user's processes
`-c`	Display only the name of the executable file and not the whole pathname
`-d`	Display the hierarchical structure for processes, showing parent–child and sibling relationships between processes using indentation
`-e`	Display the environmental information for each process
`-j`	Display for each process information about the following keywords: `user`, `pid`, `ppid`, `pgid`, `sid`, `jobc`, `state`, `tt`, `time`, and `command`.
`-l`	Display for each process information about the following keywords: `uid`, `pid`, `ppid`, `cpu`, `pri`, `nice`, `vsz`, `rss`, `mwchan`, `state`, `tt`, `time`, and `command`.
`-m`	Display information sorted according to memory usage (highest usage first)
`-o`	Similar to –O, except that it does not show the default fields and you can change header texts for multiple using multiple –o options. If no headers are specified with keywords, the header line is not displayed.
`-p`	Display information about the processes specified in the list of PIDs
`-r`	Display information sorted according to CPU usage (highest usage first)
`-u`	Display for each process information about the following keywords: `user`, `pid`, `%cpu`, `%mem`, `vsz`, `rss`, `tt`, `state`, `start`, `time`, and `command`.
`-v`	Display for each process information about the following keywords: `user`, `pid`, `state`, `time`, `sl`, `re`, `pagein`, `vsz`, `rss`, `lim`, `tsiz`, `%cpu`, `%mem`, and `command`.
`-x`	Display information about processes that do not have controlling terminals (including daemons)

The output of the `ps` command is sorted first by the terminals associated with processes and then by their PIDs. You can change the default sort order by using different options. If multiple such options are specified, the command exercises the last option. The shell sessions in this section demonstrate the use of the `ps` command with and without options. We ran all of them on a PC-BSD machine running the C shell.

The output of the PC-BSD version of the `ps` command, as shown in the following session, displays five fields about processes, whose attributes are displayed one per line: process ID (`PID`), the terminal the process is attached to (`TT`), process state (`STAT`), the CPU time the process has consumed (TIME), and the command used by the user to run the process (COMMAND). The output shows that two processes are attached to terminal 1: -csh (the login C shell) and ps, and two are attached to terminal 2: -csh (the login C shell) and vim text editor. The - in front of a shell, as in -csh, indicates that it is the login shell. The PIDs of the processes attached to terminal 1, -csh and ps, are 37838 and 41626, and have run for 19 seconds and 1 second each, respectively. Similarly, the PIDs of the processes running on terminal 2, -csh and vim, are 41496 and 41619, and each has run for 27 and 19 seconds each, respectively.

```
% ps
  PID TT  STAT    TIME COMMAND
37838  1  Ss   0:00.19 -csh (csh)
41626  1  R+   0:00.01 ps
41496  2  Ss   0:00.27 -csh (csh)
41619  2  S+   0:00.25 vim canleave
%
```

The state of a process is displayed as a character string—for example, Ss, S+, and R+ in the previous session. Table 10.2 explains the various characters in the string listed under the process state (STAT) column.

Thus, the process with the Ss state is a session leader process that is currently sleeping—that is, waiting for less than 20 seconds. Similarly, the process with the S+ state is a foreground process that has been waiting for less than 20 seconds. Finally, the process with the

TABLE 10.2 A Brief Description of the UNIX Process States Displayed by the ps Command under PC-BSD

First Character	Description	Additional Character	Description
D	Process on a disk or other short-term interruptible wait	+	Foreground process
I	Idle process—waiting for > 20 s	<	Process has raised scheduling priority
L	Process waiting for a lock	E	Exiting process
R	Runnable process waiting to get CPU	J	Process in a “jail”
S	Sleeping process—waiting for < 20 s	L	Process locked in core—for example, for raw I/O
T	Stopped/suspended process	N	Process has reduced CPU-scheduling priority
W	Idle interrupt process/thread	s	Session leader process—for example, login shell
Z	Zombie process	V	Suspended during a vfork(2)
		W	Swapped-out process
		X	Process is being traced or debugged

R+ state is a foreground process that is ready to run and is waiting to be scheduled to use the CPU. We discuss foreground and background processes in detail in Section 10.6.

The ps –u command shows the long listing of all the processes belonging to the user running the command. VSZ is the virtual size of the process and RSS (resident set size) is the real size of the process in memory. Both sizes are in kilobytes. Note that **sarwar** has logged in on five different terminals with five login C shells running.

```
% ps -u
USER      PID %CPU %MEM   VSZ  RSS TT  STAT STARTED    TIME COMMAND
sarwar 37838  0.0  0.1 25548 3900  1  Is+   6:15AM 0:00.20 -csh
(csh)
sarwar 41937  0.0  0.1 25548 3900  3  Is+   7:06AM 0:00.20 -csh
(csh)
sarwar 41985  0.0  0.1 25548 3576  4  Is+   7:07AM 0:00.10 -csh
(csh)
sarwar 44387  0.0  0.1 25548 3940  5  Ss    7:38AM 0:00.50 -csh
(csh)
sarwar 44716  0.0  0.1 16592 2304  5  R+    7:41AM 0:00.02 ps -u
sarwar 44428  0.0  0.1 25548 3900  6  Is    7:38AM 0:00.18 -csh
(csh)
sarwar 44675  0.0  0.1 16992 2800  6  I+    7:40AM 0:00.06 /bin/sh
/usr/bin/man
sarwar 44688  0.0  0.1 14788 2632  6  S+    7:40AM 0:00.03 more
%
```

You can specify a comma-separated list of UIDs to display the same information about the processes belonging to them, as in ps –u 1004,1009,1020. As a reminder, UID is the third field of a line in the **/etc/passwd** file. Note that the state of the first three processes is Is+, which means that they are foreground idle session leader processes. In other words, they are foreground login shell processes that have not used the CPU for over 20 seconds.

You can use the -U option on PC-BSD and Solaris to display the default status of all the processes belonging to the users whose comma-separated usernames are specified after the option. In the following example, all the processes belonging to the user **sarwar** are displayed. The ps –U john,davis,goldman,ibraheem command displays the default status of all the processes belonging to the users **john**, **davis**, **goldman**, and **ibraheem**.

```
% ps -U sarwar
  PID TT  STAT    TIME COMMAND
 4964  -  I    0:00.04 sshd: sarwar@pts/1 (sshd)
 8980  -  S    0:00.29 sshd: sarwar@pts/2 (sshd)
 4967  1  Is+  0:00.17 -csh (csh)
 8983  2  Ss   0:00.37 -csh (csh)
 13011  2  R+   0:00.02 ps -U sarwar
%
```

You can also use the comma-separated list of UIDs of the users to display the same information. Thus, the `ps -U 1004` command displays the default status of all the processes belonging to the user with UID 1004. Similarly, the `ps –U 1004,1005,1001` command displays the default status of all the processes belonging to the users with UIDs 1001, 1004, and 1005.

You can use the `–a` and `-x` options to display information about your processes, processes belonging to other users, system processes, and processes not attached to any controlling terminals, such as daemons. You can determine the total number of processes, including all system processes and daemons, running on a PC-BSD system by using the `ps –ax | wc –l` command. The output contains a header line and a line each for the `ps –ax` and `wc –l` processes. Thus, 130 processes are currently running on our PC-BSD system if we do not count the two processes created due to the command line itself.

```
% ps -ax | wc -l
     133
%
```

You can determine the number of processes running on a Solaris machine by using the `ps -e | wc -l` command.

You can use the `–H` option to display the kernel-visible threads, known as *lightweight processes* (LWPs) in Solaris, in the processes running on your system. The output of the `ps –axH | wc –l` command shows 506 as its output. Out of these 506 lines, one line is for the output header and one line each for the `ps –axH` and `wc –l` commands each. Thus, 503 kernel-visible threads are running in the 130 processes that are currently running on the system.

```
% ps -axH | wc -l
     506
%
```

Whereas the `–j`, `-l`, `-u`, and `-v` options allow you to display the values of predetermined keywords in a known order and the standard header information, the `–O` and `–o` options allow you to display customized output: values of the keywords of your choice, in the order of your liking, and with the header of your taste. In the following session, we show the use of the `ps` command with these options. The keywords `dsiz`, `ssiz`, and `tsiz` are, respectively, the sizes (in kilobytes) of the data, stack, and text/code segments of the *memory image* of the process. `wchan` is the event a process is waiting for. The last command in the sessions shows how you can customize the header of the output.

```
% ps -O dsiz,ssiz,tsiz,vsz,rss,wchan
  PID DSIZ SSIZ TSIZ    VSZ  RSS  WCHAN TT  STAT    TIME COMMAND
```

```
34860  172  128  348  25548  3940 pause  1  Ss   0:00.28 -csh
(csh)
35024   16  128   32  16592  2304 -      1  R+   0:00.00 ps -O
dsiz,ssiz,tsiz,vsz
29019  172  128  348  25548  3976 ttyin  4  Is+  0:00.31 -csh (csh)
% ps -o dsiz,ssiz,tsiz,lwp,nlwp,wchan,pcpu,pmem,flags,args
DSIZ SSIZ TSIZ     LWP NLWP WCHAN %CPU %MEM        F COMMAND
 172  128  348  100733     1 pause  0.0  0.1 10004002 -csh (csh)
  16  128   32  100622     1 -      0.0  0.1 10004002 ps -o
dsiz,ssiz,tsiz,lwp,nl
 172  128   348  100690    1 ttyin  0.0  0.1 10004002 -csh (csh)
% ps -o user,pid=SON -o ppid=MOM -o args
USER     SON   MOM COMMAND
sarwar 34860 34857 -csh (csh)
sarwar 35281 34860 ps -o user,pid=SON -o ppid=MOM -o args
sarwar 29019 29016 -csh (csh)
%
```

Some of the fields and the corresponding keywords are obvious and we have discussed a few of them earlier in this chapter. However, some keywords, which we will discuss now, are new and not obvious. You can display all keywords by using the ps –L command, as shown:

```
% ps -L
%cpu %mem acflag acflg args blocked caught class comm command cow
cpu cputime dsiz egid egroup emul etime etimes euid f fib flags
gid group  ignored inblk inblock jid jobc ktrace label lim
lockname login logname lstart lwp majflt minflt msgrcv msgsnd
mwchan ni nice nivcsw nlwp nsignals nsigs nswap nvcsw nwchan oublk
oublock paddr pagein pcpu pending pgid pid pmem ppid pri re rgid
rgroup rss rtprio ruid ruser sid sig sigcatch sigignore sigmask sl
ssiz start stat state svgid svuid systime tdaddr tdev tdnam time
tpgid tsid tsiz tt tty ucomm uid upr uprocp user usertime usrpri
vsize vsz wchan xstat
%
```

Table 10.3 shows some of the commonly used keywords and their meanings. Note that these fields are displayed in uppercase in the header of the output of the ps command (e.g., COMMAND for command and PID for pid).

The data, stack, and text/code segments are part of the memory image of a process. Three additional sections of the memory image of a UNIX process are: shared libraries, heap, and environment. Figure 10.6 shows the memory image of a UNIX process. The environment consists of command line arguments and five shell variables (HOME, PATH, SHELL, USER, and LOGNAME), accessible through pointers to two arrays of pointers to null-terminated strings. The stack grows from high memory to low memory and the heap

TABLE 10.3 A Brief Description of the Commonly Displayed Fields/Keywords of the `ps` Command

Field	Description	Field	Description
`command`	Command executed to create process	`ppid`	Parent's process ID
`dsiz`	Data size in kilobytes	`pri`	Scheduling priority
`cpu`	Short-term CPU usage for scheduling	`rss`	Real process size (kilobytes) in memory
`flags`	Flags indicating process states/events	`sid`	Session ID; PID of the session leader
`lwp`	Lightweight process (thread) ID	`ssiz`	Stack size in kilobytes
`majflt`	Total page faults (same as `PAGEIN`)	`started`	Time the process was started
`mwchan`	Event or lock of the locked/ blocked process	`stat`	Process state
`nlwp`	Number of threads tied to an LWP	`time`	Time the process has executed for
`pcpu`	CPU utilization up to one previous minute	`tsiz`	Text (code) size in kilobytes
`pgid`	Process group ID	`tt`	Terminal the process is attached to
`pid`	Process ID	`vsz`	Virtual size of the process in kilobytes
`pmem`	Percentage of memory used by process	`wchan`	Event (or address) on which a process waits

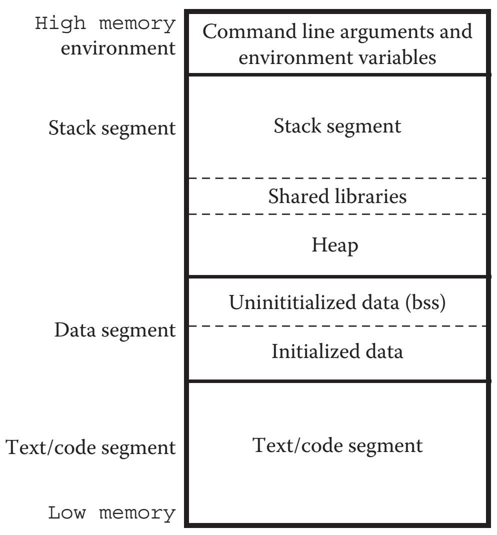


FIGURE 10.6 Memory image of a UNIX process.

grows from low memory to high memory. Text and initialized data portions are read from the program file by the `exec(2)` system call, and the uninitialized data (`bss`) section is initialized to zero, also by `exec(2)`.

You can also use the `size` command to display the sizes (in bytes) of text/code, initialized data, and uninitialized data (`bss`) for an executable program. The following are the sample runs of the command on PC-BSD and Solaris, respectively, with **/usr/bin/sort** as the argument. In both cases, the fourth number is the total size of the text and data sections (i.e., the sum of first three numbers) in decimal. For the PC-BSD version, the fifth number is the total size in hexadecimal.

```
% size /usr/bin/sort
   text     data      bss  dec    hexfilename
  52748     2656     468860092    eabc/usr/bin/sort
%
$ size /usr/bin/sort
69657  + 1584 + 44012 = 115253
$
```

The following syntax box gives a brief description of the Solaris version of the `ps` command.

SYNTAX

`ps [options]`

Commonly used display options/features:

`-G`	Display information about processes with the group IDs (GIDs) given in the comma-separated list of GIDs; no space before or after the comma
`-L`	Display processes with the number of LWPs (i.e., threads) in each process
`-P`	Display the number of the processor (`PSR`) on which an LWP is bound (i.e., executed on)
`-c`	Display information about processes according to their priorities in the kernel and user groups; the `sched` process is always listed at the top
`-d`	Display all processes, except the session leaders (the login shells for user processes and the `sched` process for the kernel processes)
`-e`	Same as the `–A` option: display information about all processes, including session leaders
`-f`	Display a full listing, including process owner ID (`UID`), process ID (`PID`), parent process ID (`PPID`), `C` (obsolete), process start time (`STIME`), `TTY`, time executed for (`TIME`), full command line (`CMD`)
`-l`	Display a long listing: state (`S`), `UID`, `PID`, `PPID`, `C` (obsolete), `PRI`, `NI`, `ADDR`, `SZ`, `WCHAN`, `TTY`, `TIME`, `CMD`
`-t`	Display a list of processes associated with a terminal; for example, `console`, `pst/4`, `term/a`, and so on
`-u` **`or`** `-U`	Display a list of processes belonging to UIDs or lognames listed in the comma-separated list

The Solaris version of the ps command supports the following options of the PC-BSD version of the command: S, a, e, r, v, w, and x. However, you invoke them without using a hyphen before them—for example, ps r. Similarly, the -o option also works under Solaris as it does under PC-BSD, but not with all the keywords listed in Table 10.3 or those displayed by the ps -L command on PC-BSD. It works with the following keywords, most of which are common with the ps version on PC-BSD: user, ruser, group, rgroup, uid, ruid, gid, rgid, pid, ppid, pgid, sid, tasked, ctid, pri, opri, pcpu, pmem, vsz, rss, osz, nice, class, time, etime, stime, zone, zoneid, f, s, c, lwp, nlwp, psr, tty, addr, wchan, fname, comm, args, projid, project, pset, and lgrp.

The output of the ps command on Solaris displays four fields, as shown, where pts/1 is pseudo terminal 1:

```
$ ps
  PID TTY         TIME CMD
 7243 pts/1       0:00 bash
 7247 pts/1       0:00 ps
$
```

The sample runs in the following session show the use of the ps command with various options. Note that the output of the ps -l command includes the obsolete fields F and ADDR, and the old-style (obsolete) PRI (known as opri in the list of keywords shown in the previous paragraph), where lower PRI value means higher priority. You can use the -y and -l options together to exclude the columns for the obsolete fields F and ADDR, include the RSS column, and display the PRI value in vogue.

```
$ ps -l
 F S   UID   PID  PPID C PRI NI  ADDR    SZ  WCHAN   TTY TIME CMD
 0 O  1007  9784  9624 0  40 20     ?  2333        pts/2 0:00 ps
 0 S  1007  9624  9621 0  50 20     ?  2546      ? pts/2 0:00 bash
$ ps -ly
   S   UID   PID  PPID C PRI NI   RSS    SZ  WCHAN   TTY TIME CMD
   S  1007 14081 14080 0  50 20  2284 10188      ? pts/5 0:00 bash
   O  1007 14116 14081 0  40 20  1472  9332        pts/5 0:00 ps
$ pagesize
4096
$
```

The first column, S, shows the state of the process. Process states in Solaris are briefly described in Table 10.4, along with some fields that are different from the fields displayed by the ps command on PC-BSD. Note that the ps command is in the running state and bash is waiting for an event; the event in this case is the termination of its child (i.e., the ps command running in foreground). NI shows the nice value of the process; the

TABLE 10.4 Brief Description of Some Fields in the Output of the `ps` Command on Solaris

Field	Meaning
`CLS`	Scheduling class: **`FSS`** Fair share scheduling **`FX`** Fixed **`IA`** Interactive **`RT`** Real-time **`SYS`** System **`SDC`** System duty cycle **`TS`** Time-sharing
`LTIME`	Execution time for a reported LWP
`NI`	Nice value: The nice value of a process; another parameter used in the computation of a process's priority value
`S`	Process state: **`O`** Currently running on a CPU (or a core) **`R`** Ready to run but not scheduled yet **`S`** Sleeping for an event to complete **`T`** Stopped: background, suspended, or being traced **`W`** Waiting for the CPU usage to drop below the upper water mark **`Z`** Zombie process (finished/terminated but is still using some kernel resources)
`STIME`	Process start time
`SZ`	Size of the process in virtual memory in pages. You can use the `pagesize` command to determine the page size on your system.
`TTY`	Terminal: Shows the name of the terminal a process is attached to

higher the value, the lower the priority of the process. You can display the priorities of processes by using the `-c` option. The `SZ` field shows the size (in pages) of the process in virtual memory. You can display the page size on your system by running the `pagesize` command. As shown previously, the page size on our system is 4096 (i.e., 4k) bytes. We have discussed the rest of the fields in the output of the `ps -ly` command earlier in this section.

You can appreciate the difference between the old-style and new-style displays of priorities by running the following command. The first column for `PRI` is for the new style and the second column is for the obsolete `PRI`. Note the priority values displayed for the two processes in the obsolete and new styles.

```
$ ps -c -o pid,pri,opri,args
  PID PRI PRI COMMAND
20041  59  40  ps -c -o pid,pri,opri,args
20033  49  50  -bash
$
```

You can use the `-t` option to display processes associated with a particular terminal, as shown in the following session. The `ps -eLP` command displays all the LWPs running in each process as well as the CPU (shown under PSR) on which a thread runs, CPU 0 in our case. Note that sched has only one thread and the zpool-rp process that processes all ZFS requests has 141 kernel-visible threads running on our system, as shown in the output of the `ps -eL | grep "zpool" | wc –l` command. The output of the `ps –eL | wc –l` command shows that 416 LWPs are currently running on the system. The last command in the session shows how you can use the –o option to display various attributes about all the processes running on the system.

```
$ ps -t pts/2
  PID TTY         TIME CMD
 9624 pts/2       0:00 bash
$ ps -eLP
  PID   LWP PSR TTY        LTIME CMD
    0     1   0 ?           0:01 sched
    5     1   0 ?           0:00 zpool-rp
    5     2   0 ?           0:00 zpool-rp
    5     3   0 ?           0:00 zpool-rp
    5     4   0 ?           0:00 zpool-rp
    5     5   0 ?           0:00 zpool-rp
...
14079     1   0 ?           0:00 sshd
14069     1   0 pts/3       0:00 man
14076     1   0 pts/3       0:00 less
14105     1   0 pts/5       0:00 ps
$ ps -eL | grep "zpool" | wc -l
     141
$ ps -eL | wc –l
     416
$ ps -e -o user,uid,pid,ppid,pri,nice,vsz,rss,nlwp,args
    USER   UID   PID  PPID PRI NI  VSZ  RSS NLWP COMMAND
    root     0     0     0  96 SY    0    0    1 sched
    root     0     5     0  99 SD    0    0  141 zpool-rpool
    root     0     6     0  99 SD    0    0    1 kmem_task
    root     0     1     0  59 20 3084 1964    1 /usr/sbin/init
    root     0     2     0  98 SY    0    0    2 pageout
    root     0     3     0  60 SY    0    0    1 fsflush
    root     0     7     0  60 SY    0    0    1 intrd
    root     0     8     0  60 SY    0    0    5 vmtasks
    root     0     9     0  99 SD    0    0    1 postwaittq
...
  sarwar  1007  9624  9621  49 20 10188 2300   1 -bash
    root     0  9620   516  59 20 16660 3892   1 /usr/lib/ssh/sshd
  sarwar  1007  9523  9522  59 20 19548 5420   1 /usr/lib/ssh/sshd
  sarwar  1007  9838  9624  59 20  9460 1556   1 ps -e -o
$
```

In Section 10.2 we discussed briefly how scheduling worked in UNIX SV. Here, we discuss how scheduling works in PC-BSD and Solaris. In PC-BSD (i.e., FreeBSD), the priority values of threads range between 0 and 255. Threshold priority is set to 120. Priorities of user processes in the system are calculated every four clock ticks (typically 40 milliseconds) using the following formula:

$$\text{priority value} = \text{threshold priority} + \text{nice value} + \left(\text{recent CPU usage}/2\right).$$

Priority value less than threshold priority is set to 120. This calculation causes the priority of a process to decrease linearly based on recent CPU usage. A digital decay filter (i.e., the decay function) is applied every second to the recent CPU usage of every process.

Unlike the older versions of the `ps` and `top` commands that reported priorities after subtracting 84 from them, the newer versions of these commands report the priority values of threads as they are stored in the kernel data structures associated with them. Table 10.5 shows the priority classes assigned to the different categories of processes/threads. Higher priority value means lower priority. The ITHD class is for threads that handle interrupts.

The priorities of the lower-half kernel threads and real-time user threads are fixed and do not change. Priorities to the kernel threads are assigned in a manner to cause minimum delays and prevent infinite blocking. Further, the lower-half (interrupt) threads may not be interrupted (preempted). Thus, the execution of threads executing under these priority classes may not be interleaved. However, depending on their recent CPU usage, the priorities of user threads may change (increase or decrease) with time in a manner similar to what has been described in Section 10.2. This means that user-level threads may move up and down between queues associated with different priorities. For this reason, we say that the UNIX uses *multilevel feedback queue scheduling*. Multiple threads in the highest priority queue are scheduled on an FCFS basis.

CPU scheduling in Solaris works quite differently and there are multiple schedulers running simultaneously. Globally, there are 170 priority values; 0 is the lowest and 169 is the highest. Higher priority value means higher priority. Unlike FreeBSD and most other UNIX systems, every thread, including a kernel thread, is preemptible and a higher priority thread may preempt a lower priority thread. This allows high-priority real-time threads

TABLE 10.5 Assignment of Priority Ranges and Priority Classes to Various Categories of Processes/Threads in FreeBSD

Process/Thread Category	**Priority Range**	**Priority Class**
Lower-half kernel (interrupt)	0–47	`ITHD`
Real-time user	48–79	`REALTIME`
Top-half kernel	80–119	`KERN`
Time-sharing user	120–223	`TIMESHARE`
Idle user	224–255	`IDLE`

TABLE 10.6 Assignment of Priority Ranges and Priority Classes to Various Categories of Processes/Threads in Solaris

Process/Thread Category	**Priority Range**	**Priority Class**
Lower-half kernel (interrupt)	160+	
Real-time	100–159	`RT`
System/kernel	60–99	`SYS`
Interactive, time-sharing, fixed, fair share	0–59	`IA, TS, FX, FSS`

to preempt kernel threads. Thus, in a single-processor system, no kernel or time-sharing process runs while a runnable real-time process exists in the system. Table 10.6 shows the priority classes assigned to different categories of processes/threads in Solaris.

In the following in-chapter exercises, you will use the `ps` command with and without options to appreciate the command output.

EXERCISE 10.1

Use the `ps` command to display the status of processes that are running in your current session. Can you identify your login shell? What is it?

EXERCISE 10.2

Run the command to display the status of all the processes running on your system. What command did you run? What are their PIDs? What are the PIDs of the parents of all the processes?

10.5.2 Dynamic Display of Process Attributes

If you want to monitor the CPU activity and attributes of processes in real time, you can use the `top` command. It displays the status of the most CPU-intensive processes on the system, by default, in terms of the following process attributes: PID, owner's logname, number of threads in a process, process priority, nice value of the process, size of the process in virtual memory, resident part of the process (i.e., current size of the process in the main memory), process state, amount of time a process has executed for, percentage of CPU time used by a process, and command name used to invoke the process. On PC-BSD, the command also displays the CPU number (or the core number) on which a process (or thread) runs. The `top` command continues to run until you press <`q`> or <`Q`>, and keeps updating the status every second or so on PC-BSD and every five seconds on Solaris. The periodic update time can be specified through a command line option when the `top` command is executed or through an interactive command while `top` runs. The number of processes whose status is displayed depends on the size of the display screen or window size on a GUI-based system. On smart displays or windows, it is usually 15–30 lines, one per process.

The command also displays several other important system statistics in a short header, comprising 5–8 lines, displayed before the statistics of the processes and/or threads. The header information depends on the UNIX variant on which the command runs. The common information in the headers on PC-BSD and Solaris includes load average, time (days+hours:minutes:seconds) the system has been up and running, current time, number of processes on the system, number of processes sleeping, number of zombie processes, CPU utilization (percentages of idle time, user time, kernel/system time, and interrupt-handling time), utilization of user area in the main memory (total and free), and swap space (total and free). The Solaris version additionally displays the following numbers on a per-second basis: context switches, traps, interrupts, system calls, page faults, memory (in kilobytes) swapped in and out, and the number of children processes created through the `fork(2)` and `vfork(2)` system calls (discussed in Chapter 19). The PC-BSD version also displays the last-used PID and the virtual memory (demand paging) information, including the number of most frequently used (MFU) and least recently used (LRU) pages. The header of the `top` command on Mac OS X (Darwin) also includes information about the resident (in-memory) shared libraries, page faults, incoming and outgoing network traffic, and disk read/write operations. Table 10.7 shows the snapshots of the header of the `top` command on PC-BSD, Solaris, and Mac OS X (Darwin).

Most of the features of the `top` command can be selected by an interactive command while `top` runs. The following syntax box and Table 10.8 give a brief description of the `top` command's interactive keystrokes on PC-BSD.

TABLE 10.7 Headers of the `top` Command for PC-BSD and Solaris

PC-BSD Version

```
last pid: 68271;  load averages:  0.05,  0.18,  0.22   up
 44+04:02:01  13:45:14
100 processes: 1 running, 98 sleeping, 1 zombie
CPU:  0.0% user,  0.0% nice,  0.3% system,  0.1% interrupt, 99.6% idle
Mem: 32M Active, 671M Inact, 1776M Wired, 1423M Free
ARC: 1232M Total, 465M MFU, 342M MRU, 16K Anon, 142M Header, 283M Other
Swap: 2048M Total, 2048M Free
```

Solaris Version

```
load averages:  0.00,  0.00,  0.00;              up 4+05:56:55 18:51:51
56 processes: 55 sleeping, 1 on cpu
CPU states: 99.9% idle,  0.0% user,  0.1% kernel,  0.0% iowait,  0.0% swap
Kernel: 222 ctxsw, 523 intr, 51 syscall
Memory: 4060M phys mem, 2980M free mem, 1024M total swap, 1024M free swap
```

MacOS X (Linux Darwin) Version

```
Processes: 75 total, 2 running, 73 sleeping, 372 threads        13:50:38
Load Avg: 0.26, 0.23, 0.18  CPU usage: 0.97% user, 1.45% sys, 97.57% idle
SharedLibs: 4164K resident, 15M data, 0B linkedit.
MemRegions: 8959 total, 680M resident, 21M private, 439M shared.
PhysMem: 1220M wired, 965M active, 1142M inactive, 3327M used, 4863M free.
VM: 158G vsize, 1043M framework vsize, 127404(0) pageins, 0(0) pageouts.
Networks: packets: 21688/7786K in, 14229/3289K out.
Disks: 68678/2474M read, 26190/2661M written.
```

TABLE 10.8 Brief Description of the Interactive Keystrokes of `top` Under PC-BSD

Command	Meaning
`H`	Display/toggle the threads statistics on separate lines
`P`	Display/toggle CPU usage statistics on per-CPU basis
`S`	Display/toggle system processes
`^L`	Redraw screen
`h or ?`	Display summary of interactive keystrokes
`k`	Kill processes whose space-separated PID-list is specified
`n or #`	Change the number of processes to display
`o`	Sort the display by `cpu`, `res`, `size`, `time`, and so on
`r`	Change the nice value of a list of processes specified as space-separated PIDs
`s`	Change periodic update time (in seconds)
`t`	Toggle the display of `top` process
`z`	Toggle the display of idle processes

SYNTAX

`top [options]`

Purpose: Display and periodically update information about processes currently running on the system

Output: The various attributes of processes currently running on the system, periodically updated

Commonly used options/features:

`N`	Display status of N processes
`-H`	Display statistics for each thread of a multithreaded process, one line per thread
`-I`	Do not display idle processes
`-P`	Display CPU usage statistics on a per-CPU basis
`-S`	Display/toggle system processes
`-U username`	Display statistics for only those processes belonging to `username`
`-a`	Display full command line for each process
`-o field`	Sort display by `field`, a header of a column in output in lowercase—for example, `cpu`, `pri`, `res`, or `time`
`-s`	time Update screen every `time` seconds; default is 5
`-t`	Do not display the `top` process
`-z`	Do not display idle system processes

The options `-a`, `-H`, `-I`, `-P`, `-S`, `-t`, and `-z` are really toggles.

The `top` command on Solaris supports all of the features that its PC-BSD counterpart supports. In most cases, even the options and interactive keystrokes are the same. In a

TABLE 10.9 Brief Description of the Interactive Keystrokes of `top` Under Solaris

Command	Meaning
`H or t`	Toggle the display of threads on separate lines.
`M`	Sort the display by memory usage (as if the `top -o size` command was executed)
`N`	Sort the display by PID (as if the `top -o pid` command was executed)
`P`	Sort the display by CPU usage (as if the `top -o cpu` command was executed)
`T`	Sort the display by CPU time (as if the `top -o time` command was executed)
`h or ?`	Display summary of interactive keystrokes
`k`	Kill processes whose space-separated PID-list is specified
`n or #`	Change the number of processes to display
`r`	Change the nice value of a list of processes specified as space-separated PIDs
`s`	Change periodic update time (in seconds)

few cases, the interactive command strokes and option letters are different. The following syntax box and Table 10.9 give a brief description of the interactive keystrokes of the `top` command on Solaris.

SYNTAX

`top [options]`

Commonly used options/features:

The `N`, `-I`, `-S`, `-U`, `-o`, and `-s` options are common between the PC-BSD and Solaris versions of `top`.

`-a` Display statistics for all processes to the extent possible

`-c` Display full command line for each process

`-t` Display statistics for each thread of a multithreaded process, one line per thread

We show some example sessions of the `top` command primarily under PC-BSD because most of the options and interactive keystrokes under the two systems are common. Almost all of the uncommon options and interactive keystrokes use different letters, but provide the same features. The following is a run of the command without any options. The command output shows that the system has been up and running for almost 33 days without crashing. The last PID assigned is 75597. Further, 106 processes are currently in the system, with one running, 104 sleeping, and one in the zombie state. At the top-right corner is the current time that is updated when `top` updates its output. The MySQL daemon (msqld), owned by the user **david**, is the second-highest priority process after `top`. It has 41 threads, has a size of 207M bytes out of which 43.552M bytes is in the main memory, and has run

TABLE 10.10 Brief Description of Various Fields of the Output of the `top` Command

Field	Meaning
`up`	System up time: Total time (days+hours:minutes:seconds) the system has been running for without going down
`CPU`	CPU utilization: Percentage of user, nice, system, interrupt processing, and idle times
`Mem`	Physical memory: Active, inactive, wired (including BIO-level data and code), cache, buf (number of bytes used for BIO-level disk caching), free
`ARC`	Adaptive replacement cache (ARC): A page replacement algorithm with better performance than the least recently used (LRU) algorithm, MRU (most recently used) bytes, MFU (most frequently used) bytes, anon (being delivered) bytes, header bytes, and other miscellaneous bytes
`Swap`	Swap space: Total set aside and free (unused) at this time
`PID`	Process ID: ID of the process
`USERNAME`	Username of the process owner
`THR`	Threads: Number of threads in the process
`PRI`	Priority: Priority value of a process that dictates when the process is scheduled; the smaller the priority value of a process, the higher its priority
`NICE`	Nice value: The nice value of a process; another parameter used in the computation of a process's priority value. The range of this value is −20–19 (20 on Solaris and some other UNIX systems).
`SIZE`	Size: The size of the memory image of a process (text, data, and stack) in kilobytes
`RES`	Resident memory: Current amount of process memory in kilobytes that resides in physical memory
`STATE`	Process state: Current state of the process—for example, start, sleep, run (or CPUn, such as CPU2, on symmetric multiprocessor [SMP] systems), idl, zomb, stop, wait, lock—or the current event on which the processes waits for—for example, select (waiting for the `select` system call to return)
`C`	CPU number: CPU number on which the process runs—for example, 2 means CPU2. Numbering starts with 0
`TIME`	Process running time: Amount of time (minutes:seconds) a process has run for, excluding waiting time in the ready queue or waiting for an I/O device
`WCPU`	Weighted CPU usage: Recent CPU utilization, a parameter used in computing a process's priority for scheduling purposes
`COMMAND`	Command: Lists the command used to start this process. The `-f` option is needed to see the full command in System V UNIX; otherwise, only the last component of the pathname is displayed.

for 55 minutes and 45 seconds. You can appreciate other values in the header and attributes of the processes by using Table 10.10. If you observe the command output for a little while, you will notice that the processes move up and down as their priorities change. Similarly, the statistics in the header are also updated periodically.

```
% top
last pid: 75597;  load averages:  0.19,  0.25,  0.22   up
32+21:31:09  07:14:21
106 processes: 1 running, 104 sleeping, 1 zombie
CPU: 0.1% user, 0.0% nice, 2.6% system, 0.2% interrupt, 97.1% idle
Mem: 38M Active, 662M Inact, 1287M Wired, 592K Cache, 1914M Free
```

```
ARC: 931M Total, 374M MFU, 331M MRU, 16K Anon, 139M Header, 86M
                                                           Other
Swap: 2048M Total, 2048M Free

  PID USERNAME    THR PRI NICE   SIZE    RES STATE   C   TIME
                                                    WCPU   COMMAND
75567 sarwar        1  20    0 19772K  2852K CPU2    2   0:00
                                                       0.98%  top
2355 david         41  20    0   207M 43552K sbwait  1  55:45
                                                      0.00% mysqld
1445 root           1  20    0 14404K  1812K select  1  51:28
                                                      0.00% powerd
1808 haldaemon      2  20    0 63524K  8148K select  2  29:17
                                                        0.00% hald
1853 root           1  20    0 23264K  2668K select  1  23:43
                                            0.00% hald-addon-storage
2460 malik          6  39   19 88284K 48720K uwait   0  22:06
                                                    0.00% virtuoso-t
2200 malik          5  52    0   202M 30772K select  1  15:38
                                           0.00% pc-systemupdatertra
2339 malik          4  49    0   906M   141M select  3  14:12
                                                    0.00% kdeinit4
1496 root           1  27    0 12268K  1720K nanslp  0   9:52
                                                     0.00% swapexd
1442 root           1  20    0 25340K  3652K select  3   7:29
                                                        0.00% ntpd
1742 root           1  20    0 60832K  6068K select  0   1:38
                                                        0.00% sshd
1233 root           1  20    0 14428K  1872K select  1   1:04
                                                     0.00% syslogd
2387 root           2  25    0   192M 28116K select  3   0:55
                                                 0.00% pc-mounttray
2436 malik          1  20    0   449M 68736K select  1   0:41
                                                      0.00% korgac
1745 root           1  20    0 16524K  2072K nanslp  1   0:31
                                                        0.00% cron
1703 root           1  20    0 64328K 13632K select  1   0:29
                                                   0.00% python2.7
```

You can interact with top while it runs by using various interactive keystrokes. You can press h to display the various keystrokes that allow you to interact with top. When you use an interactive command, top prompts you with one or more questions related to the chore that you want it to perform. For example, when you press n, top prompts you for the number of processes to display. You input the number and hit the <Enter> key for top to start displaying information about said number of processes. Similarly, if you want to terminate a process, press k and top prompts you for the PID of the process to be terminated.

You input the PID of the process to be terminated and hit `<Enter>` for `top` to terminate the process. So, if you want to display the real-time status of the processes owned by the user **bob**, you press `u` and enter the login name of the user. The following output shows the monitoring of **bob**'s processes.

```
...
ARC: 931M Total, 374M MFU, 331M MRU, 16K Anon, 139M Header, 86M
Other
Swap: 2048M Total, 2048M Free
Username to show: bob
  PID USERNAME THR PRI NICE   SIZE    RES STATE   C  TIME   WCPU
                                                           COMMAND
78800  bob        1  20    0 19772K  2876K CPU0    0  0:01   0.59%
                                                               top
73317  bob        1  36    0 25548K  3904K pause   3  0:00   0.00%
                                                               csh
78671  bob        1  20    0 86100K  6948K select  3  0:00   0.00%
                                                              sshd
73450  bob        1  21    0 14788K  2632K ttyin   2  0:00   0.00%
                                                              more
```

If you want `top` to display the CPU statistics on a per-CPU basis, you press `P`. The following output shows the statistics of the four CPUs (or cores) in your system, CPU0–CPU3.

```
last pid: 80939;  load averages:  0.25,  0.29,  0.28   up
32+22:35:07  08:18:19
116 processes: 1 running, 114 sleeping, 1 zombie
CPU 0:  0.4% user,  0.0% nice,  0.7% system,  0.0% interrupt,
                                                      98.9% idle
CPU 1:  0.4% user,  0.0% nice,  8.8% system,  0.0% interrupt,
                                                      90.8% idle
CPU 2:  0.0% user,  0.0% nice,  0.7% system,  0.4% interrupt,
                                                      98.9% idle
CPU 3:  0.0% user,  0.0% nice,  0.7% system,  0.0% interrupt,
                                                      99.3% idle
Mem: 47M Active, 663M Inact, 1289M Wired, 592K Cache, 1902M Free
ARC: 931M Total, 374M MFU, 331M MRU, 16K Anon, 139M Header, 86M
                                                             Other
Swap: 2048M Total, 2048M Free
<P>
  PID USERNAME  THR PRI NICE   SIZE    RES STATE   C   TIME  WCPU
                                                           COMMAND
80921  sarwar     1  20    0 19772K  3032K CPU1    1   0:00  0.78%
                                                               top
 2355 malik       41  20    0   207M 43552K sbwait  1 55:50  0.00%
                                                            mysqld
 1445 root         1  20    0 14404K  1812K select  3 51:33   0.00%
                                                             powerd
 1808 haldaemon    2  20    0 63524K  8148K select  2 29:20   0.00%
                                                               hald
 1853 root         1  20    0 23264K  2668K select  3 23:45   0.00%
                                                          hald-addo
...
```

In the following session, we show that you can display system processes by pressing `S`. Note that the `kernel` process has 170 kernel-visible threads. The `idle` process runs when the system has no process to run. The system comes out of the idle state when an interrupt occurs—for example, a user presses a key on the keyboard or the clock ticks.

```
...
ARC: 931M Total, 374M MFU, 331M MRU, 16K Anon, 139M Header, 86M
                                                              Other
Swap: 2048M Total, 2048M Free
<S>
  PID USERNAME THR PRI NICE   SIZE    RES STATE   C   TIME   WCPU
                                                            COMMAND
   11 root        4 155 ki31     0K    64K CPU3    3 3129.1 400.00%
                                                               idle
80921  sarwar    1  20    0 19772K  3032K CPU0    0   0:03   0.59%
                                                                top
   21 root        1  16    -     0K    16K syncer  3 106:34   0.29%
                                                             syncer
   12 root       24 -84    -     0K   384K WAIT    3 384:39   0.00%
                                                               intr
    0 root      170   8    0     0K  2720K -       3 307:24   0.00%
                                                             kernel
    4 root        5  -8    -     0K    96K tx->tx  2  72:46   0.00%
                                                            zfskern
   14 root        1 -16    -     0K    16K -       2  64:36   0.00%
                                                          rand_harv
...
```

We now show a sample run of `top` on a Solaris machine. As you can see, the system has been up and running for about six days and currently has 59 processes with 1 using the CPU and the remaining 58 sleeping (i.e., waiting for some event to occur). Further, the following kernel events happened during the last one second: 215 context switches, 187 traps, 500 interrupts, 342 system calls, and 149 page faults. The CPU, main memory, and swap space statistics are displayed in a manner similar to the output of the `top` command on PC-BSD. Finally, note that `NLWP` in the third column of the statistics about currently running processes stands for the number of LWPs, and is equivalent to `THD` (threads) shown in the output of the `top` command on PC-BSD. The Service Management Facility (SVC) master starter daemon (svc.startd) and the SVC configuration daemon (svc.configd) have 12 and 23 LWPs associated with them, respectively.

```
$ top
load averages:  0.00,  0.00,  0.00;       up 5+23:32:07   12:27:03
59 processes: 58 sleeping, 1 on cpu
CPU states: 99.9% idle,  0.0% user,  0.2% kernel,  0.0%
                                                iowait,  0.0% swap
Kernel: 215 ctxsw, 187 trap, 500 intr, 342 syscall, 149 flt
Memory: 4060M phys mem, 2970M free mem, 1024M total swap, 1024M
                                                      free swap
   PID USERNAME NLWP PRI NICE  SIZE   RES STATE    TIME    CPU
                                                        COMMAND
 15538 sarwar      1  59    0   10M 2292K sleep    0:00  0.01%
                                                           bash
 15548 sarwar      1  59    0 4208K 2460K cpu/1    0:00  0.01% top
 15537 sarwar      1  59    0   19M 5432K sleep    0:00  0.01%
                                                           sshd
  1215 smmsp       1  59    0 7020K 1472K sleep    0:00  0.00%
                                                       sendmail
    13 root       12  59    0   37M   29M sleep    0:06  0.00%
                                                    svc.startd
    86 root        1  59    0 9760K 1060K sleep    0:03  0.00%
                                                      in.mpathd
 15543 sarwar      1  59    0   19M 5448K sleep    0:00  0.00%
                                                           sshd
   567 root       28  59    0   98M   16M sleep    0:06  0.00% fmd
    47 root        9  59    0 4332K 2792K sleep    0:02  0.00%
                                                        dlmgmtd
   845 root        4  59    0 3228K 1408K sleep    0:02  0.00%
                                                    devchassisd
    15 root       23  59    0   20M   19M sleep    0:18  0.00%
                                                    svc.configd
    76 netadm      6  59    0 4476K 2768K sleep    0:07  0.00%
                                                        ipmgmtd
   213 root        6  59    0   13M 3676K sleep    0:02  0.00%
                                                       devfsadm
  1217 root        1  59    0 7016K 1840K sleep    0:03  0.00%
                                                       sendmail
...
```

You can use the interactive keystrokes listed in Table 10.8 and Table 10.9 to observe their effect on the command display.

EXERCISE 10.3

Try the previous sessions for the `top` command on your system. How many processes are running on your system? What are the priority and nice values for the highest priority process?

10.6 PROCESS AND JOB CONTROL

UNIX is responsible for several activities related to process and job management, including process creation, process termination, running processes in the foreground and background, suspending and resuming processes, and switching processes from foreground to background and vice versa. As a UNIX user, you can request the process and job control tasks by using the shell commands discussed in this section.

10.6.1 Foreground and Background Processes and Related Commands

When you type a command and hit `<Enter>`, the shell executes the command and returns by displaying the shell prompt. While your command executes, you do not have access to your shell and therefore cannot execute any commands (i.e., continue work) until the current command finishes and the shell returns. When commands execute in this manner, we say that they execute in the *foreground*. By default, every process runs in the foreground, taking input from the keyboard and sending output to the display screen.

UNIX allows you to run a command so that, while the command executes, you get the shell prompt back and can submit other commands. This capability is called running the command in the *background*. You can run a command in the background by ending the command with an ampersand (&). Of course, in a graphical environment, you can run a command in one terminal window in the foreground, open another terminal window, and use the shell in the new terminal window. However, this activity takes time and consumes additional system resources.

Background processes run at lower priorities compared to their foreground counterparts. Thus, they get to use the CPU only when no higher priority process needs it. When a background process generates output that is sent to the display screen, the screen looks garbled, but if you are simultaneously using another application, your work is not altered in any way. You can get out of the application and then get back into it to obtain a cleaner screen. Some applications such as vim allow you to redraw the screen without quitting it. In vim (see Chapter 3), pressing `<Ctrl+L>` in Command mode allows you to do so.

The syntaxes for executing commands in the foreground and background are as follows. Note that no space is needed between the command and & but that you can use space for clarity.

SYNTAX

```
command (for foreground execution)
command & (for background execution)
```

Now consider the following command executed under the Bourne shell. It searches the whole file structure for a file called **foo** and stores the pathnames of the directories that contain this file in the file **foo.paths**. The error messages are sent to the file **/dev/null**, which is the UNIX black hole: whatever goes in never comes out. Note that, for the C shell, 2> should be replaced with >&. This command may take several minutes, perhaps hours, depending on the size of the file structure, system load (in terms of the number of users logged on), and the number of processes running on the system. So if you want to do some other work on the system while the command executes, you cannot do so because the command executes in the foreground.

```
$ find / -name foo -print > foo.paths 2> /dev/null
...
$
```

The find command is a perfect candidate for background execution because, while it runs, you have access to the shell and can do other work. Thus, the preceding command should be executed as follows:

```
$ find / -name foo -print > foo.paths 2> /dev/null &
[1] 23467
$
```

The number shown in brackets is returned by the shell and is the job number (also called job ID) for the process; the other number is the PID of the process. Here, the job number for the find command is 1 and the PID is 23467. A job is a process that is not running in the foreground and is accessible only at the terminal with which it is associated. Such processes typically run in the back or are suspended processes.

The commands that perform tasks that do not involve user intervention and take a long time to finish are good candidates for background execution. Some examples are sorting large files (sort command), compiling large programs (cc, gcc, CC, make, etc.), computationally intensive programs such as one that determines if a large integer number is prime, and searching a large file structure for one or more files (find command). Commands that do terminal I/O, such as the vim editor, are, of course, not good candidates for background execution. The reason is that, when such a command executes in the background, it stops accepting input from the keyboard. The command needs to be brought back to the foreground before it can start running again. The fg command allows you to bring a background process to the foreground.

While running a command in the foreground, you might need to *suspend* it in order to go back to the shell, do something under the shell, and then return to the suspended process. For example, say that you are in the middle of editing a C program file with vim and need to compile the program to determine whether some errors have been corrected. You can save changes to the file, suspend vim, compile the program to view the results of the compilation, and return to vim. You can suspend a foreground process by pressing <Ctrl+Z>, move a suspended process to the foreground by using the fg command, and move a suspended process to the background by using the bg command. So you can suspend vim by pressing <Ctrl+Z>, compile the program to identify any other errors, and resume the suspended vim session by using the fg command.

SYNTAX

```
fg [%jobid]
bg [%jobid-list]
```

Purpose: Syntax 1: Resume execution of the process with job number jobid in the foreground or move background processes to the foreground; a jobid starts with %

Syntax 2: Resume execution of suspended processes/jobs with job numbers in `jobid-list` in the background; a `jobid` starts with %

Commonly used options/features:

`% or %+`	Current job
`%-`	Previous job
`%N`	Job number **N**
`%Name`	Job beginning with **Name**
`%?Name`	Command containing **Name**

If there are multiple suspended processes, the `fg` command without an argument brings the current process into the foreground, and the `bg` command without an argument resumes execution of the current process in the background. The job using the CPU at any particular time is called the *current job*.

You can use the `jobs` command to display the job numbers of all suspended (stopped) and background processes and identify which one is the current process. The current process is identified by a + and the previous process by a `-` in the output of the `jobs` command. The following is a brief description of the command.

SYNTAX

```
jobs [option] [%jobid-list]
```

Purpose: Display the status of the suspended and background processes specified in `jobid-list`; with no list, display the status of current job

Commonly used options/features:

`-l` Also display PIDs of jobs

In the following sessions, we show the use of the `fg`, `bg`, `<Ctrl+Z>`, and `jobs` commands. We run the `sort` and `cp` commands in the background. The jobID and PID pairs of these processes are [1] 60149 and [2] 60156 for the `sort` and `cp` commands, respectively. The `sort bigdata` command is in the currently running process/job, as shown in the output of the `jobs` and `jobs –l` commands, as well as the `R` state of the process in the output of the `ps` command. The `cp bigdata bigdata1` command has been swapped on the disk, as indicated by the D state of the process in the output of the `ps` command.

```
% sort bigdata > bigdata.sorted &
[1] 60149
```

```
% cp bigdata bigdata1 &
[2] 60156
% ps
  PID TT  STAT    TIME COMMAND
56222  1  Ss   0:00.67 -csh (csh)
60149  1  R    0:07.94 sort bigdata
60156  1  S    0:00.00 -csh (csh)
60157  1  D    0:00.34 cp bigdata bigdata1
60164  1  R+   0:00.00 ps
% jobs
[1]  + Running               sort bigdata > bigdata.sorted
[2]  - Running               cp bigdata bigdata1
% jobs -l
[1]  + 60149 Running               sort bigdata > bigdata.sorted
[2]  - 60156 Running               cp bigdata bigdata1
%
```

In the following session, the first `fg` command brings the current job into the foreground. The `fg %2` command brings job number 2 into the foreground. A string that uniquely identifies a job can also be used in place of a job number. The string is enclosed in double quotes if it has spaces in it. The third `fg` command illustrates this convention. The `jobs –l` command, as expected, shows both jobs as suspended. The output of the `ps` command shows the state of the `cp bigdata bigdata1` command as `TW`, which means that the process has been suspended and swapped out to disk temporarily. We use the `bg %1 %2` command to start the background execution of the two suspended processes. We later confirm this status by using the `jobs` command, which shows both processes as running, with the `sort` process currently using the CPU. We also show that the `bg` command without argument puts the current (or the only suspended) process into the background. The `ps –p 60149` command shows that the `sort` process is in the running state and has been in the system for 11 minutes and 3.78 seconds.

```
% fg
sort bigdata > bigdata.sorted
<Ctrl+Z>
Suspended
% fg %2
cp bigdata bigdata1
<Ctrl+Z>
Suspended
% fg %"sort"
sort bigdata > bigdata.sorted
<Ctrl+Z>
Suspended
% jobs -l
[1]  + 60149 Suspended             sort bigdata > bigdata.sorted
[2]  - 60156 Suspended             cp bigdata bigdata1
```

```
% ps
  PID TT  STAT    TIME COMMAND
56222  1  Ss   0:00.96 -csh (csh)
60149  1  T    0:29.29 sort bigdata
60156  1  TW   0:00.00 -csh (csh)
60157  1  TW   0:00.00 cp bigdata bigdata1
61261  1  R+   0:00.02 ps
% bg %1 %2
[1]    sort bigdata > bigdata.sorted &
[2]    cp bigdata bigdata1 &
% jobs
[1]  + Running              sort bigdata > bigdata.sorted
[2]  - Running              cp bigdata bigdata1
% fg %1
sort bigdata > bigdata.sorted
<Ctrl+Z>
Suspended
% bg
[1]    sort bigdata > bigdata.sorted &
% jobs
[1]  + Running              sort bigdata > bigdata.sorted
[2]  - Running              cp bigdata bigdata1
% ps -p 60149
  PID TT  STAT     TIME COMMAND
60149  2  R    11:03.78 sort bigdata
%
```

As discussed earlier in this section, we discuss the following session as an additional example to explain the working of the `fg` and `<Ctrl+Z>` commands. The `gcc -o lab8 lab8.c` command is used to compile the C program in the **lab8.c** file and put the executable in a file called **lab8**. Understanding what compilation means is not the point here, and a fuller discussion of the syntax and semantics of the `gcc` command is presented in Chapter 17. Here, we are merely emphasizing that processes that take a long time to start or those that have executed for a considerable amount of time are usually good candidates for processes to be suspended. The example of suspending the `vim` command is presented only as an illustration. This sequence of events is shown in the following session. Note that the output of the `ps` command after `vim` has been suspended shows the status of `vim` as `T`, which means that vim has been suspended/stopped.

```
% ps
PID TT  STAT    TIME COMMAND
587  1  Ss   0:00.17 -csh (csh)
616  1  R+   0:00.00 ps
% vim lab8.c
#include <stdio.h>
#define SIZE 100
```

```
int main (int argc, char *argv[])
{
...
<Ctrl+Z>
Suspended
% ps
  PID TT  STAT     TIME COMMAND
  587  1  Ss    0:00.41 -csh (csh)
  812  1  T     0:00.13 vim lab8.c
 1035  1  R+    0:00.02 ps
% gcc -o lab8 lab8.c
% fg %1
#include <stdio.h>
#define SIZE 100
main (int argc, char *argv[])
{
...
:q!
%
```

In the following in-chapter exercise, you will practice the creation and management of foreground and background processes by using the `bg`, `fg`, and `jobs` commands.

EXERCISE 10.4

Run the sessions presented in this section on your system to practice foreground and background process creation and switching processes from the foreground to the background (with the `bg` command) and vice versa (with the `fg` command). Use the `jobs` command to display the job IDs of the active and suspended processes.

10.6.2 UNIX Daemons

Although any process running in the background can be called a *daemon*, in UNIX jargon a daemon is a system process running in the background. Daemons are frequently used in UNIX to offer various types of services to users or running software and handle system administration tasks. For example, printing, logging, e-mail, web browsing, remote login via Secure Shell, file transfer, interaction through social networking sites, and finger services are provided via daemons. Printing services are provided by the printer daemon lpd. Finger services (see Chapter 11) are handled by the finger daemon fingerd. The inetd daemon, commonly known as the UNIX superserver, handles various Internet-related services by spawning the relevant server daemons at system boot time. Access the **/etc/inetd.conf** file to view the services offered by this daemon on your system. This file has one line for every service that inetd offers.

10.6.3 Sequential and Parallel Execution of Commands

You can type multiple commands on one command line for the sequential and/or parallel execution of these commands. The following is a brief description of the syntax for sequential execution of commands specified in one command line.

SYNTAX

`cmd1; cmd2; …; cmdN`

Purpose: Execute commands **`cmd1, cmd2, …, cmdN`** sequentially as separate processes

Note that the semicolon is used as a command separator and, therefore, does not follow the last command. No spaces are needed before and after a semicolon, but you can use spaces for clarity. These commands execute one after the other, as though each were typed on a separate line. In the following session, the `date` and `echo` commands execute sequentially as separate processes. The first session is on PC-BSD running C shell and the second is on Solaris running Bash. Note the difference between the outputs of the `date` command on the two systems.

```
% date; echo Hello, World!
Sat Sep 20 10:27:50 PKT 2014
Hello, World!
%
$ date; echo Hello, World!
Saturday, September 20, 2014 10:29:05 PM PKT
Hello, World!
$
```

You can specify parallel execution of commands in a command line by ending each command with an ampersand (&). The commands that end with `&` also execute in the background. No spaces are required before or after `&`, but you can use spaces for clarity. When you execute a command in the background, the shell displays the following pair as output: `[jobID]  PID`, where the first job ID is 1, increasing linearly by adding one to the last-used job ID. Like PIDs, a job ID may be recycled if it has not been assigned to a process currently. The following is a brief description of the syntax for parallel execution of shell commands specified in one command line.

SYNTAX

`cmd1& cmd2& … cmdN&`

Purpose: Execute commands **`cmd1, cmd2, …, cmdN`** in parallel as separate processes

The following sessions were executed on PC-BSD under C shell and Solaris under Bash, respectively. The `date` and `echo` commands execute in parallel and in the background, followed by the sequential execution of the `uname` and `who` commands in the foreground. In general, since the `who` command executes at the end, its output is always displayed at the end. The outputs of the other three commands (date, echo, and uname) may be displayed in any order. This order is due to the scheduling of processes and the amount of time each takes to execute. Thus, the same output order may or may not be reproduced if you execute the command line again. In the following session, the outputs are displayed in the order echo, uname, date, and who for PC-BSD, and echo, date, uname, and who for Solaris. The job and process ID pairs of the date and echo commands are [1] 15167 and [2] 15168, respectively, on PC-BSD. Similarly, the job and process IDs for the date and echo commands on Solaris are [1] 6802 and [2] 6803, respectively.

```
% date& echo Hello, World!& uname; who
[1] 15167
[2] 15168
Hello, World!
FreeBSD
Sat Sep 20 10:31:51 PKT 2014
malik           pts/0       Aug 14 09:45 (:0)
sarwar          pts/1       Sep 20 10:27 (static-
host202-147-168-98.link.net.pk)
[2]  + Done                         echo Hello, World!
[1]  + Done                         date
%
```

```
$ date& echo Hello, World\!& uname; who
[1] 6802
[2] 6803
Hello, World!
Saturday, September 20, 2014 03:35:57 PM PKT
SunOS
[1]-  Done                  date
[2]+  Done                  echo Hello, World\!
root       console      Sep 19 17:13
sarwar     pts/1        Sep 20 15:30    (static-
host202-147-168-98.link.net.pk)
$
```

The last & in a command line puts all the commands since the previous & in one process. In the following command line, therefore, the date command executes as one process and all the commands in who; whoami; uname; echo Hello, World!& as another process. The job and process ID pairs for these processes are [1] 49380 and [2] 49381, respectively, on PC-BSD.

```
% date & who ; whoami ; uname ; echo Hello, World! &
[1] 49380
[2] 49381
[sarwar@pcbsd-srv] ~/unix3e/ch10% Sat Sep 20 17:49:58 PKT 2014
malik           pts/0       Aug 14 09:45 (:0)
sarwar          pts/1       Sep 20 17:45 (182.178.199.200)
```

```
malik            pts/2        Sep 20 10:59 (:0)
sarwar
FreeBSD
Hello, World!
<Enter>
[2]    Done                  ( who; whoami; uname; echo Hello, World! )
[1]   + Done                  date
%
```

As shown in the following session, when run on Solaris, the job ID and process ID pair for the two processes are [1] 7540 and [1] 7544, respectively. Note that whereas job IDs for the two jobs on PC-BSD are 1 and 2, they are 1 each on Solaris. Further, the process IDs for the two processes on PC-BSD are two consecutive numbers, whereas they are nonconsecutive on Solaris. This means that under PC-BSD, these processes were created right after each other and were assigned consecutive job IDs and consecutive process IDs. On the other hand, after the creation of the date process, three other processes with process IDs 7541, 7542, and 7543 were created before the second process in our command line was created. This explains both of our processes getting job ID 1 each. Note that we replace `'!` in the `echo` command with `\!` in Bash under Solaris. Finally, as shown in the shaded regions of the two sessions, since the shell prompt returns on both systems after the two processes (jobs) have been created but before any or some output has been displayed on the screen, we have to hit the `<Enter>` key to display the shell prompt again.

```
$ date & who; whoami; uname; echo Hello, World\! &
[1] 7540
Saturday, September 20, 2014 10:54:27 PM PKT
root       console      Sep 19 17:13
sarwar     pts/1        Sep 20 22:48    (182.178.199.200)
[1]+  Done                        date
sarwar
SunOS
[1] 7544
$ Hello, World!
<Enter>
[1]+  Done                        echo Hello, World\!
$
```

As will be discussed briefly in Chapter 12, UNIX allows you to group commands and execute them as one process by separating commands using semicolons and enclosing them in parentheses. This is called *command grouping*. Because all the commands in a command group execute as a single process, they are executed by the same subshell. However, all the commands execute sequentially, one after the other. The following is a brief description of the syntax for command grouping.

SYNTAX

(cmd1; cmd2; …; cmdN)

Purpose: Execute commands **cmd1, cmd2, …, cmdN** sequentially, but as one process

In the following session, therefore, the `date` and `echo` commands execute sequentially, but as one process.

```
% (date; echo Hello, World!)
Sat Sep 20 22:14:52 PKT 2014
Hello, World!
%
```

You can combine command grouping with sequential execution by separating command groups with other commands or command groups. In the following session, the `date` and `echo` commands execute as one process, followed by the `who` command executing as a separate process.

```
% (date; echo Hello, World!); who
Sun Sep 21 08:05:55 PKT 2014
Hello, World!
root            pts/0        Aug 14 09:45 (:0)
sarwar          pts/1        Sep 21 08:05 (39.59.18.169)
malik           pts/2        Sep 20 10:59 (:0)
%
```

Command groups can be nested. Hence, `((date; echo Hello, World!); who)` and `((date; echo Hello, World!); (who; uname))` are valid commands and produce the expected results. Command grouping makes more sense when groups are executed as separate processes, as shown in the following session.

```
% (date ; echo Hello, World!)&
[1] 40563
% Sun Sep 21 08:08:42 PKT 2014
Hello, World!
[1]    Done                         ( date; echo Hello, World! )
% (date; echo Hello, World)& (who; uname)& whoami
[1] 40936
[2] 40938
Sun Sep 21 08:12:30 PKT 2014
Hello, World
malik           pts/0        Aug 14 09:45 (:0)
sarwar          pts/1        Sep 21 08:05 (39.59.18.169)
```

```
malik           pts/2        Sep 20 10:59 (:0)
sarwar
[1]  - Done                          ( date; echo Hello, World )
FreeBSD
% <Enter>
[2]    Done                          ( who; uname )
%
```

In the second group of commands, `(date; echo Hello, World)` and `(who; uname)` execute in the background and the `whoami` command executes in the foreground; all three commands execute in parallel. Again, the order of output is dependent on the scheduling of these commands.

In the following in-chapter exercises, you will practice sequential and parallel execution of UNIX commands.

EXERCISE 10.5

Run the sessions presented in this section on your system to practice sequential and parallel execution of shell commands.

EXERCISE 10.6

Which of the following commands run sequentially and which run in parallel? How many of the processes run in parallel? `(who; date) & (cat temp; uname & whoami)`

10.6.4 Abnormal Termination of Commands and Processes

When you run a command, it terminates normally after successfully completing its task. A command (process) can terminate prematurely because of a bad argument that you passed to it, such as a directory argument as source file to the `cp` command or because of a run-time error. At times, you might also need to terminate a process abnormally. The need for *abnormal termination* arises when you run a process with a legal but wrong argument (e.g., a wrong file name to the `find` command) or when a command is taking too long to finish, perhaps, due to an infinite loop. Here, we address abnormal termination in relation to both foreground and background processes.

You can terminate a foreground process by pressing `<Ctrl+C>` or using the `kill` command from another shell. You can terminate a background process in one of two ways: (1) by using the `kill` command, or (2) by first bringing the process into the foreground by using the `fg` command and then pressing `<Ctrl+C>`. The primary purpose of the `kill` command is to send a *signal* (also known as a *software interrupt*) to a process. The UNIX operating system uses a signal to get the attention of a process. You can send any one of the several signal types supported by your UNIX system to a process that you own or you have the permission to do so. A process can take one of three actions upon receiving a signal:

1. Accept the default action as determined by the UNIX kernel
2. Ignore the signal
3. Intercept the signal and take a user-defined action

For most signals, the default action, in addition to some other events, always results in termination of the process. Ignoring a signal does not have any impact on the process. A user-defined action is specified as a program statement (usually a function call) that takes control to a specific piece of code in the process. In a shell script, you can specify these actions by using the `trap` command in the Bourne shell. The C shell provides a limited handling of signals via the `onintr` command. In a C program, you can specify these actions by using the library call `signal`. We discuss the `trap` and `onintr` commands in detail in Chapters 13 and 15, respectively. We describe the `signal(2)` system call in detail in Chapter 19. For a quick look, view its manual page by using the `man signal` or `man 3 signal` command on PC-BSD or Solaris system. The `man –S3 signal` command can also be used on PC-BSD.

Signals can be generated for various reasons. The processes themselves cause some of these reasons, whereas others are external to processes. A signal caused by an event internal to a process is known as an *internal signal*, or a *trap* (not to be confused with the `trap` command in the Bourne shell). For example, the execution of a divide-by-zero instruction in a process generates a trap. A signal caused by an event external to a process or a hardware device in the computer system is called an *external signal*. If an external signal is for a hardware device such as a CPU or disk controller, it is called a *hardware interrupt* or *interrupt*. An external event meant to get attention of one or more processes is known as a *software interrupt* or *signal* in the UNIX jargon. For example, an internal signal is generated for a process when the process tries to execute a non-existing instruction or access a memory region that it is not allowed to access such as memory belonging to some other process or the UNIX kernel. You can generate an external signal by pressing `<Ctrl+C>`, by logging out, or by using the `kill` command. The `kill` command can be used to send any type of signal to a process. The following is a brief description of the `kill` command.

SYNTAX

```
kill [-s signal_name] proc-list
kill [-signal_name] proc-list
kill [-signal_number] proc-list
kill -l [exit_status]
```

Purpose: Syntaxes 1–3: Send the signal for `signal _ number` or symbolic `signal _ name` to processes whose PIDs or jobIDs are specified in space-separated `proc-list`; jobIDs must start with `%`.

Syntax 4: The command `kill -l` returns a list of all the signals along with their numbers and names (Solaris); on PC-BSD, only symbolic names without the `SIG` prefix are displayed without numbers. The operand `exit _ status` specifies a signal number or the exit status of a terminated or completed process.

Commonly used options/features:

1	`HUP` (Hang-up)
2	`INT` (Interrupt: `<Ctrl+C>`)
3	`QUIT` (Quit: `<Ctrl+\>`)
6	`ABRT` (Abort)
9	`KILL` (Sure kill: Nonignorable, noninterceptable)
14	`ALRM` (Alarm clock)
15	`TERM` (Software termination: the default signal number)

You can use the `kill -1` (number one) command to send the default signal to all of your processes. Only a superuser can send a signal to the processes belonging to other users. Thus, a superuser can use this command to send signals to all the processes running on the system. You can use the `kill –l` (lowercase L) command to display the signals supported by your UNIX system. The following command executed on PC-BSD shows that it supports 32 signal types. The manual page for the signal system call shows that PC-BSD supports 33 signal types. You can use the `man signal` command to verify for yourself. The syntax box shows details of some of the more commonly used signals.

```
% kill -l
HUP INT QUIT ILL TRAP ABRT EMT FPE KILL BUS SEGV SYS PIPE ALRM
TERM URG STOP
TSTP CONT CHLD TTIN TTOU IO XCPU XFSZ VTALRM PROF WINCH INFO USR1
USR2 LWP
%
```

Solaris supports 72 signal types, as shown in the following session.

```
$ kill -l
 1) SIGHUP        2) SIGINT        3) SIGQUIT       4) SIGILL
 5) SIGTRAP       6) SIGABRT       7) SIGEMT        8) SIGFPE
 9) SIGKILL      10) SIGBUS       11) SIGSEGV      12) SIGSYS
13) SIGPIPE      14) SIGALRM      15) SIGTERM      16) SIGUSR1
17) SIGUSR2      18) SIGCHLD      19) SIGPWR       20) SIGWINCH
21) SIGURG       22) SIGIO        23) SIGSTOP      24) SIGTSTP
25) SIGCONT      26) SIGTTIN      27) SIGTTOU      28) SIGVTALRM
29) SIGPROF      30) SIGXCPU      31) SIGXFSZ      32) SIGWAITING
33) SIGLWP       34) SIGFREEZE    35) SIGTHAW      36) SIGCANCEL
37) SIGLOST      38) SIGXRES      39) SIGJVM1      40) SIGJVM2
41) SIGRTMIN     42) SIGRTMIN+1   43) SIGRTMIN+2   44) SIGRTMIN+3
45) SIGRTMIN+4   46) SIGRTMIN+5   47) SIGRTMIN+6   48) SIGRTMIN+7
```

```
49) SIGRTMIN+8   50) SIGRTMIN+9   51) SIGRTMIN+10  52) SIGRTMIN+11
53) SIGRTMIN+12  54) SIGRTMIN+13  55) SIGRTMIN+14  56) SIGRTMIN+15
57) SIGRTMAX-15  58) SIGRTMAX-14  59) SIGRTMAX-13  60) SIGRTMAX-12
61) SIGRTMAX-11  62) SIGRTMAX-10  63) SIGRTMAX-9   64) SIGRTMAX-8
65) SIGRTMAX-7   66) SIGRTMAX-6   67) SIGRTMAX-5   68) SIGRTMAX-4
69) SIGRTMAX-3   70) SIGRTMAX-2   71) SIGRTMAX-1   72) SIGRTMAX
%
```

Table 10.11 shows some examples of the `kill` command and their meanings.

The *hang-up* signal is generated when you log out, the *interrupt* signal is generated when you press `<Ctrl+C>`, and the *quit* signal is generated when you press `<Ctrl+\`. The `kill` command sends signal number 15 to the process whose PID is specified as an argument. The default action for this signal is termination of the process that receives it. This signal can be intercepted and ignored by a process, as can most of the other signals. In order to terminate a process that ignores signal 15 or other signals, signal number 9, known as *sure kill*, has to be sent to it. The `kill` command terminates all the processes whose PIDs are given in the **proc-list**, provided that these processes belong to the user who is using `kill`. The following session on PC-BSD presents some instances of how the `kill` command can be used. But don't kill the login shell process because it will log you out.

```
% jobs -l
[1]  + 14338 Running                sort bigdata > bigdata.sorted
[2]  - 14667 Running                cp biggerdata biggerdata.bak
[3]    14668 Running                grep sh biggerdata > lines.sh
% kill 14668
% kill -2 14667
[3]    Terminated                   grep sh biggerdata > lines.sh
% kill -9 14338
[2]    Interrupt                    cp biggerdata biggerdata.bak
% jobs
```

TABLE 10.11 Some Examples of the `kill` Command and Their Meanings

Command	Meaning
`kill 1234`	Send the default signal (`SIGTERM`) to the process with PID 1234

`kill -9 1234`
`kill -s kill 1234`
`kill -s KILL 1234`
	Send `SIGKILL` (guaranteed termination signal) to the process with PID 1234

`kill -9 1234 -1004`
`kill -s kill 1234 -1004`
`kill -s KILL 1234 -1004`
	Send `SIGKILL` (guaranteed termination signal) to the process with PID 1234 and to all the processes with process group ID (PGID) 1004

`kill -TERM -1004`
`kill -- -1004`
	Send `SIGTEM` to all the processes with process group ID 1004

```
[1]    Killed                          sort bigdata > bigdata.sorted
% jobs
%
```

In the first case, the kill command sends signal number 15 to a process with PID 14668. In the second case, signal number 2 (`SIGINT`) is sent to a process with PID 14667. In both cases, because the specified signal numbers are not intercepted, the processes are terminated. The `kill` command can be used to terminate a number of processes with one command line. For example, the command `kill -9 13586 20581` terminates processes with PIDs 13586 and 20581.

Process ID 0 can be used to refer to all the processes created during the current login. Thus, the `kill -9 0` command terminates all processes resulting from the current login, as is shown in the following session. Note that the command has terminated all the processes resulting from **sarwar**'s current login session, including the process that maintains the Secure Shell login connection with the client. Thus, you see the prompt of the local machine, a terminal window running Bash on a MacBook Pro under Mac OS X (Darwin), `MacBook-Pro:~ syedsarwar$`. This has serious consequences because the order in which `kill` terminates processes is not known in this case. Thus, if the login shell and the process that maintains the Secure Shell connection with the client machine are terminated first, then the processes executing under the login shell continue to run. You would notice this when you login again and realize that the `ssh` command is taking longer than usual to establish the connection with the remote machine, and once you are connected and have a login shell running, system response time is poor. The `ps` command would show that your background processes from the previous login are still running, as shown in the following session in the shaded portions. Note that the statuses of these "leftover" processes are `D` (swapped out) and `DL` (swapped out and locked), and they are not associated with any terminal (note the negative terminal numbers). You need to terminate these processes in order to improve the system response time, as shown in the last line of the session.

```
% ps -U sarwar
PID TT  STAT    TIME COMMAND
15361  -  S    0:00.01 sshd: sarwar@pts/1 (sshd)
15368  1  Ss   0:00.07 -csh (csh)
15424  1  D    0:07.05 sort bigdata
15437  1  D    0:00.60 cp biggerdata biggerdata.bak
15456  1  D    0:01.31 grep sh biggerdata
15521  1  R+   0:00.00 ps -U sarwar
15158  2- DL   1:58.99 sort bigdata
% kill -9 0
Connection to 192.102.169.10 closed.
MacBook-Pro:~ syedsarwar$ ssh sarwar@192.102.169.10
Password for sarwar@pcbsd-srv:
Last login: Sun Oct  5 10:17:45 2014 from 139.15.192.135
```

```
FreeBSD 10.0-RELEASE-p6 (GENERIC) #0 acf484b(releng/10.0): Mon Feb
24 15:14:38 EST 2014
Welcome to FreeBSD!
...
% ps
  PID TT  STAT    TIME COMMAND
15424  1- DL   0:23.56 sort bigdata
15437  1- D    0:09.15 cp biggerdata biggerdata.bak
15456  1- D    0:21.06 grep sh biggerdata
15158  2- DL   1:59.38 sort bigdata
15866  3  Ss   0:00.06 -csh (csh)
15913  3  R+   0:00.00 ps
% kill 15158 15424 15437 15456
%
```

The `kill` command also works with job numbers. Hence, the following command can be used to terminate a process with job number 1. You can terminate multiple processes by specifying their job numbers in the command line. For example, `kill -9 %1 %3` can be used to terminate processes with job numbers 1 and 3.

```
$ kill -9 %1
[1] + Killed find / -name foo -print > foo.paths &
$
```

When you log out, all the processes running in your session get a hang-up signal (signal number 1) and are terminated per the default action. If you want processes to continue to run even after you have logged out, you need to execute them so that they ignore the hang-up signal when they receive it. You can use the UNIX command `nohup` to accomplish this task. The following is a brief description of the syntax for this command.

SYNTAX
`nohup command [args]`

Purpose: Execute **`command`** and ignore the hang-up signal

You need to use the `nohup` command for processes that take a long time to finish, such as a program sorting a large file containing hundreds of thousands of customer records. Obviously, you would run this type of program in the background so that it runs at a lower priority. The following is a simple illustration of the use of the `nohup` command. Here, the `find` command runs in the background and is not terminated when you log out or send it signal number 1 (hang-up) via the `kill` command. If output of the command is not redirected, it is appended to the **nohup.out** file by default.

```
$ nohup find / -name foo -print 1> foo.paths 2> /dev/null &
[1] 62808
$ kill -1 62808
$ jobs
[1]+  Running   nohup find / -name foo -print > foo.paths 2> /dev/
null &
$ kill 62808
$ <Enter>
[1]+  Terminated    nohup find / -name foo -print > foo.paths 2>
/dev/null
$ jobs
$
```

If you separate commands with semicolons, you can run them with nohup. In the following session, GenData generates some data and puts it in a file called **employees**, and the sort command sorts the file and stores the sorted version in the **employees.sorted** file.

```
$ nohup GenData > employees ; sort employees > employees.sorted &
[2] 15931
$
```

In the following in-chapter exercises, you will use the kill command to practice abnormal termination of processes, and the nohup and ps -a commands to appreciate how you can run processes that do not terminate when you log out.

EXERCISE 10.7

Give a command for terminating processes with PID 10974 and jobID 3.

EXERCISE 10.8

Run the first of the nohup commands, use ps to verify that the command is executing, log out, log in again, and use the ps -a command to determine whether the find command is running.

10.7 PROCESS HIERARCHY IN UNIX

When you turn on your UNIX system, the kernel—after performing some checks and other household tasks—creates the first process from scratch. In PC-BSD, this process is called kernel and in Solaris it is called sched. This process, which has no parent, has PID 0. The PPID of this process is also 0. It then spawns several other processes, which are meant to handle several important kernel tasks, including tasks related to virtual memory, file handling, interrupt handling, and performing various tasks at boot time such as

TABLE 10.12 Some of the Key Processes Created by `kernel` (on PC-BSD) and `sched` (on Solaris)

PC-BSD	Solaris	Purpose
`init`	`init`	Granddaddy of all user processes; all user-level services, including all Internet services, and user processes run under the children of this process.
`pagedaemon`	`pageout`	Read/write pages from/to disk
`bufdaemon`	`fsflush`	Supplies clean pages/buffers by writing dirty pages/buffers to disk
`zfskern`	`zpool-rpool`	Performs I/O tasks related to ZFS pools
`intr`	`intrd`	Handles all interrupts (lower-half of the kernel)
`vmdaemon`	`vmtasks`	Moves processes from main memory to disk storage

initializing hardware ports. A list of a few of these processes and their purpose is given in Table 10.12.

The `init` process is the granddaddy of all user processes that are created, so long as the system is up and running. The `init` process has a PID of 1 and runs with superuser privileges. The executable binary for the process is in the file **/sbin/init** under PC-BSD and in **/usr/sbin/init** under Solaris. This process, after performing several activities, as given in the **/etc/rc** file, reads the **/etc/ttys** file to determine which I/O (terminal) lines are to be active. For each active line, `init` starts a `getty` process from **/etc/getty** file. The `getty` process, also running in superuser mode, sets terminal attributes, such as baud rate, as specified in the file **/etc/termcap**. It then displays the `login:` prompt, inviting you to log on to the terminal.

At the `login:` prompt, when you type your login name and press `<Return>`, the `getty` process forks a child. The child process executes the `exec` system call to become a login process with your login name as its parameter. The login process prompts you for your password and checks the validity of your login name and password. If it finds both to be correct, the login process execs to become your login shell. If the login process does not find your login name in the **/etc/passwd** file or finds that the password that you entered does not match the one in the **/etc/passwd** file, it displays an error message and terminates. Control goes back to the `getty` process, which redisplays the `login:` prompt. Once in your login shell, you can do your work and terminate the shell by pressing `<Ctrl+D>` or running the `exit` command. When you do so, the shell process terminates and control goes back to the `getty` process, which displays the `login:` prompt, and life goes on.

When you login through the `ssh` command, the sshd daemon (the Secure Shell server process) on the remote machine creates a child sshd process for you (your private sshd) that handles communication with your client. The server sshd goes back and looks for more SSH connection requests from other clients. Your private sshd spawn another sshd process that overwrites itself with a pseudo terminal **pts/1**, which prompts you for the password. When you enter the correct password, it runs the login shell process for you. By default, the login shell is C shell on PC-BSD and Bash on Solaris. You run your commands under your login shell. When you press `<Ctrl+D>` to logout, your login shell terminates along with your private sshd daemon. You can also logout by running the `exit` command.

Figures 10.7 and 10.8 show schematically the process hierarchies on Solaris and PC-BSD, respectively, with logins on a terminal and a pseudo terminal. Two processes, `ps` and `more`, are running under login shells Bash and C shell on Solaris and PC-BSD, respectively.

Thus, when you log on to a UNIX system, the system creates the first process for you, called your *login shell*. The login shell interprets/executes your commands by creating processes for all the commands that you execute (see Section 10.3 for details of command execution).

Two UNIX processes exist throughout the lifetime of a system: `kernel` (PC-BSD) or `sched` (Solaris) and `init`. The `getty` process, which monitors a terminal line, lives for as long as the terminal is attached to the system. Your login shell process lives for as long as you are logged on. All other processes are usually short-lived and stay for as long as a command or utility executes.

You can use the `ptree` command on Solaris to display the process tree of the currently running processes on the system in a graphical form, showing the parent–child relationships. You can display the process tree for a process or for a user. In the following session, we use the `ptree -a davis` command to show the process hierarchy for the user **davis**.

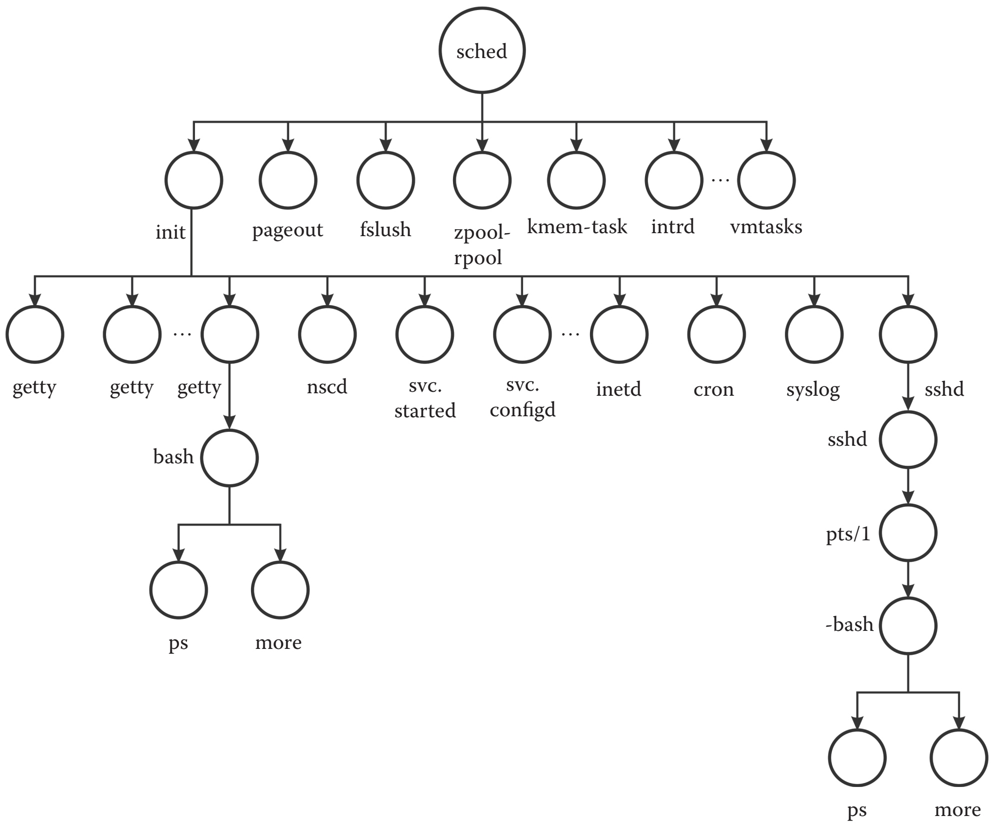


FIGURE 10.7 UNIX process hierarchy on Solaris with logins on a terminal and a pseudo terminal.

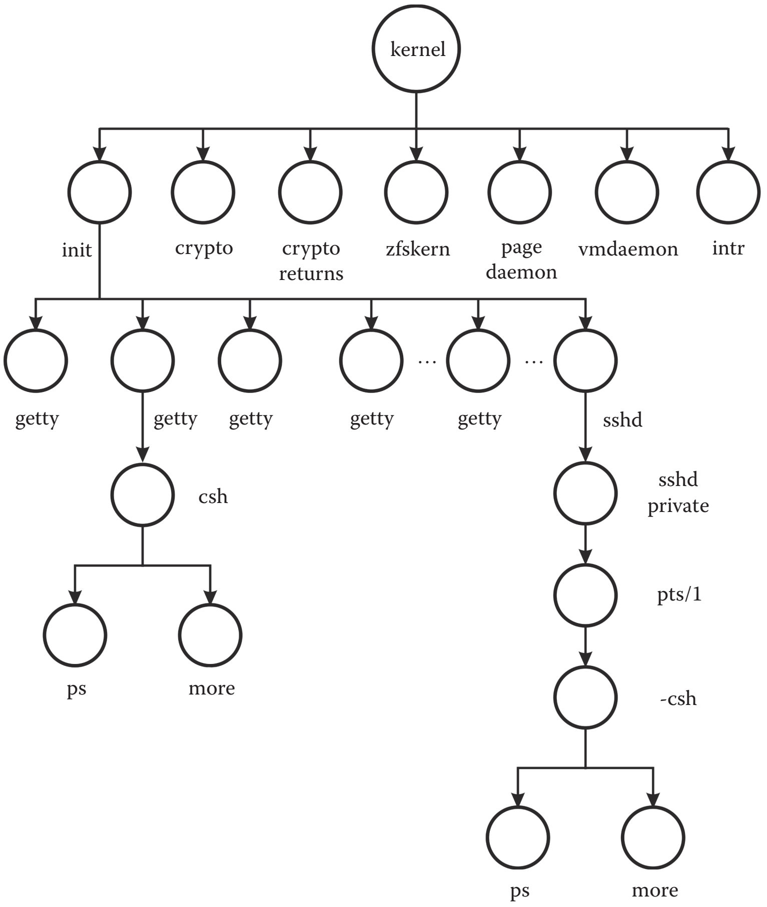


FIGURE 10.8 UNIX process hierarchy on PC-BSD with logins on a terminal and a pseudo terminal.

The output shows that the granddaddy of all of the processes run by **davis** is, as expected, the `init` process. The output further shows that **davis** is using the system through an SSH session and is currently using Bash. Without the `–a` option, the line for the `init` process, the parent of process 516 (i.e., the sshd daemon) is not displayed. The `ptree -a 'pgrep sshd'` command shows the process hierarchy for the sshd daemon.

```
$ ptree -a davis
1     /usr/sbin/init
  516   /usr/lib/ssh/sshd
    26087 /usr/lib/ssh/sshd
      26088 /usr/lib/ssh/sshd
        26095 -bash
          26188 ptree -a davis
$ ptree -a 'pgrep ssh'
1     /usr/sbin/init
  516   /usr/lib/ssh/sshd
```

```
    26087 /usr/lib/ssh/sshd
      26088 /usr/lib/ssh/sshd
        26095 -bash
          26192 ptree -a 516 26087 26088
$
```

Under PC-BSD, you can use the `ps –auxd` command to display the parent–child relationship between processes. However, the output for this command is not as complete and fancy as it is for the `ptree` command, as can be seen in the following session. You can try running the `ps -auxfd | more` command on your system to see the complete process hierarchy on your system.

```
% ps -dU sarwar
  PID TT  STAT     TIME COMMAND
93204  -  S     0:00.18 sshd: sarwar@pts/1 (sshd)
93207  1  Ss    0:00.45 - -csh (csh)
96636  1  R+    0:00.02 `-- ps -dU sarwar
%
```

On PC-BSD, you can run the `ps -aflx | more` command to display all the processes running on your system and some of their important attributes, including PID, PPID, priority, nice value, virtual size, size currently in main memory, status, event waiting for, and full command name used to start the process. You can display the usernames, PIDs, PPIDs, and full command names of all the processes running on Solaris by executing the `ps -e -o user,pid,ppid,comm` command. We used the outputs of these commands to construct the process hierarchies on the two systems, shown in Figures 10.7 and 10.8.

SUMMARY

A process is a program in execution. Being a time-sharing system, UNIX allows execution of multiple processes simultaneously. On a computer system with one CPU, processes are executed concurrently by scheduling the CPU time and giving it to each process for a short time called a quantum. Each process is assigned a priority by the UNIX system, and when the CPU is available, it is given to the process with the highest priority. The priority classes and the way priorities are assigned to various categories of processes are different on PC-BSD and Solaris.

The shell executes commands by creating child processes using the fork and exec system calls. When a process uses the fork system call, the UNIX kernel creates an exact main memory image of the process. The shell itself executes an internal command. An external binary command is executed by the child shell overwriting itself by the code of the command via an exec call. For an external command comprising a shell script, the child shell executes the commands in the script file one by one.

Every UNIX process has several attributes, including process ID (PID), process ID of the parent (PPID), process name, process state (running, suspended, swapped, zombie, etc.), the terminal the process was executed on, the length of time the process has run, and process priority. The `ps` command may be used to view a static display of these attributes. The `top` command may be used to view a dynamic display of the various system statistics and attributes of the processes running on the system, and permit interactive commands via various keystrokes.

UNIX processes can be run in the background or the foreground. A foreground process controls the terminal until it finishes, so the shell cannot be used for anything else while a foreground process runs. When a process runs in the background, it returns the shell prompt so that the user can do other work as the process executes. Because a background process runs at a lower priority, a command that takes a long time is a good candidate for background execution. The background system processes that provide various services are called daemons. A set of commands can be run in a group as separate processes or as one process. Multiple commands can be run from one command line as separate processes by using a semicolon (`;`) as the command separator; enclosed in parentheses, these commands can be executed as one process. Commands can be executed concurrently by using an ampersand (`&`) as the command separator.

Suspending processes, moving them from the foreground to the background and vice versa, having the ability to display their status, interrupting them via signals, and terminating them are all known as job control, and UNIX has a set of commands that allow these actions. Foreground processes can be suspended and moved to the background pressing `<Ctrl+Z>` followed by executing the `bg` command. Suspended and background processes can be moved to the foreground by using the `fg` command. Commands that are suspended or run in the background are also known as jobs. The `jobs` command can be used to view the status of all your jobs. You can press `<Ctrl+C>` to terminate a foreground process.

The `kill` command can terminate a process with its PID or job ID. The command can be used to send various types of signals, or software interrupts, to processes. Upon receipt of any signal except one, a process can take the default (kernel-defined) action, take a user-defined action, or ignore it. No process can ignore the `kill -9` command, the sure kill signal, which was put in place by the UNIX designers to make sure that every process running on a system could be terminated. Commands executed with the `nohup` command keep running even after a user logs out. The `kill -9 0` command is the sure kill for all the processes associated with the current login of a user.

You can use the `ptree` command on a Solaris system running Bash to display tree structures for your or some other user's processes. With the `–a` option, the output displayed contains a line for the granddaddy `init` process. Similarly, you can use the `ps –d` command under C shell on PC-BSD to see a less sophisticated version of the parent–child and sibling relationships between processes on the system. The `ps -auxfd` command displays the complete hierarchy of all the processes on the system and the `ps –d` command displays the process hierarchy for a particular user.

QUESTIONS AND PROBLEMS

1. What is a process? What is the process ID of a process?
2. What is CPU scheduling? How does a time-sharing system run multiple processes on a computer system with a single CPU? Be brief but precise.
3. Name three famous CPU-scheduling algorithms. Which are parts of the UNIX scheduling algorithm?
4. What are the main states that a process can be in? What does each state indicate about the status of the process?
5. What is the difference between built-in (internal) and external shell commands?
6. How does a UNIX shell execute built-in and external commands? Explain your answer with an example.
7. Name three process attributes.
8. What is the purpose of the `nice` command in UNIX?
9. What are foreground and background processes in UNIX? How do you run shell commands as foreground and background processes? Give an example for each.
10. In UNIX jargon, what is a daemon? Give examples of five daemons.
11. What are signals in UNIX? Give three examples of signals. What are the possible actions that a process can take upon receiving a signal? Write commands for sending these signals to a process with PID 10289.
12. Give a command that displays the status of all running processes on your system.
13. Give a command that returns the total number of processes running on your system.
14. Compute the priority number of a UNIX process with a recent CPU usage of 31, a threshold priority of 60, and a nice value of 20. Show your work.
15. Give the sequence of steps (with commands) for terminating a background process.
16. Create a zombie process on your UNIX system. Use the `ps` command to show the process and its state.
17. The `ps -auxw` or `ps auxw` command is one of the most useful commands. What does it display? Explain your answer.
18. Give two commands to run the `date` command after 10 seconds. Make use of the `sleep` command; read the relevant manual page to find out how to use it.
19. Run a command that would remind you to leave for lunch after one hour by displaying the message `Time for Lunch!`

20. Give a command for running the `find` and `sort` commands in parallel.

21. Give an example of a UNIX process that does not terminate with `<Ctrl+C>`.

22. Run the following commands on one command line so that they do not terminate when you log out. What command did you use?

```
find / -inum 23476 -print > all.links.hard 2> /dev/null
find / -name foo -print > foo.paths 2> /dev/null
```

23. Run the following sequence of commands under your shell. What are the outputs of the three `pwd` commands? Why are the outputs of the last two `pwd` commands different?

```
$ pwd
$ sh
$ cd /usr
$ pwd
...
$ <Ctrl+D>
$ pwd
...
$
```

24. Run the `top` command on your system. What are the priority and nice values of the highest priority process? Run commands to have `top` display information about the top-10 processes and refresh its output every seven seconds. What commands did you use?

25. As you monitor the top session, display processes for the user **john**. What command did you use? Show your work.

26. Use the `ptree` command to display the tree structure for the processes running in your current session. What command did you use? Which process is the grandparent of all your processes and what is its process ID? What command will you use to display the tree hierarchy for the processes that the user **kent** has run on your system?

27. What are the names of processes with process IDs 0, 1, 2, and 3 on your UNIX system? How did you get the answer to the question? Show your work.

28. Suppose you are running various programs in a session—ssh, vim, etc.—and the terminal locks up or the remote login program crashes, causing you to be disconnected from the host. Or, perhaps your keyboard or mouse suddenly stops working. Explain how you could log in from another terminal and use a sequence of UNIX commands to recover from the situation. Give the sequence of UNIX commands you would use.

29. What command would you use to display the hierarchical structure of processes on your system? What is the name of the process with PID 0? How many children does

this process have and what are their names? What is the pathname of the executable for the `init` process?

30. Write commands for displaying the number of threads in the `kernel` process (the granddaddy of all processes) and the `init` process (the granddaddy of all user processes). How long has the `kernel` process been running on your system? How did you find out?

31. The `ps -U root,bin,goldman,ibraheem` command displays the default status of all the processes belonging to the users **root**, **bin**, **goldman**, and **ibraheem**. However, the output does not show the logname of the process owner. Write down the command, for both PC-BSD and Solaris, along with a sample run on your machine that would display the username for each process.

32. What command line will you use on Solaris to display all the processes running under the priority class `SYS`? Show the results of running the command on your system. What command will you use to display the total number of processes running under the `TS` (time-sharing) priority class?

33. What is the `SDC` priority class? Give example of at least one process that runs under this priority class.

34. What is the purpose of applying the decay function to the CPU usage of processes before the priority values of processes are recalculated? Explain with an example.

CHAPTER 11

Networking and Internetworking

Objectives

- To describe networks and internetworks and explain why they are used
- To discuss briefly the TCP/IP protocol suite, IP addresses, protocol ports, and Internet services and applications
- To explain what the client–server software model is and how it works
- To discuss various network software tools for electronic communication, remote login, file transfer, remote command execution, tracing a route in the Internet, and status reporting
- To describe in detail the Secure Shell and other secure commands
- To cover the commands and primitives

  `finger, ftp, ifconfig, host, nslookup, ping, rcp, rlogin, rsh, ruptime, rwho, scp, sftp, slogin, ssh, talk, telnet, traceroute`

11.1 INTRODUCTION

The history of computer networking and the Internet goes back to the late 1960s, when the Department of Defense's Advanced Research Projects Agency (ARPA) started funding networking research. This research resulted in a wide area network, called ARPANET, by the late 1970s, with five nodes—UCLA, Stanford, UC Santa Barbara, University of Utah, and BBN. In 1982, a prototype Internet that used Transmission Control Protocol/Internet Protocol (TCP/IP) became operational and was utilized by a few academic institutions, industrial research organizations, and the US military. By early 1983, all US military sites connected to ARPANET were on the Internet, and computers on the Internet numbered 562. By 1986, this number had more than quadrupled to 2308. From then on, the size of the Internet doubled every year for the next 10 years, until it served about 9.5 million computers by 1996. The first Web browser, called Mosaic, was developed at the National Center for Supercomputer Applications (NCSA) and launched in 1991. As a result, World Wide Web (shortened to the www, or just the Web) browsing surpassed File Transfer Protocol (FTP) as the major use of the Internet by 1995. The first website, info.cern.ch, was launched on August 6, 1991. Since the second edition of this book was written, the social networking sites have had a major impact on the use of the Internet. Facebook, Flickr, YouTube, Reddit, Twitter, Tumblr, Dropbox, and Pinterest were born during this period of time. As of the writing of this book, Facebook and Twitter have over 1.59 billion and over 320 million monthly active users, respectively, generating over 5 billion likes and 500 million tweets per day.

Today, the Internet serves over 1.01 billion hosts, around 1 billion websites and over 14.7 billion webpages live on the Internet, 49.5 billion webpages are indexed by Google, over 1 yottabyte (10^{24} bytes) of data resides on the Internet, over 40% of the world's population uses the Internet (i.e., over 2.93 billion users), over 1.2 billion domain names are in use, over 200 billion e-mails are sent per day, and over 259 of 263 countries, colonies or territories, and disputed territories in the world provide access. It is projected that by 2020, over 60% (5 billion) of the planet will be connected by the Internet. UNIX has a special place in the world of networking in general and internetworking in particular because most of the networking protocols were initially implemented on UNIX platforms. Also, server processes running on UNIX-based computers provide most of the Internet services.

11.2 COMPUTER NETWORKS AND INTERNETWORKS

When two or more computer hardware resources (computers, printers, scanners, plotters, etc.) are connected, they form a computer *network*. A hardware resource on a network or an internetwork is usually referred to as a *host*. Figure 11.1 (a) shows a schematic diagram of a network with six hosts, **H1–H6**.

Computer networks are categorized as *local area networks* (LANs), *metropolitan area networks* (MANs), and *wide area networks* (WANs), based on the maximum distance between two hosts on the network. Networks that connect hosts in a room, building, or buildings of a campus are called LANs. The distance between hosts on a LAN can be anywhere from a few meters to about one kilometer. Networks that are used to connect hosts within a city, or between small cities, are known as MANs. The distance between hosts on a MAN is about 1 to 20 km. Networks that are used to connect hosts within a state or country are known as WANs. WANs are also known as *long-haul networks*. The distance between the hosts on a WAN is in the range of tens of kilometers to a few thousand kilometers.

An *internetwork* is a network of networks. Internetworks can be used to connect networks within a campus or networks that are thousands of kilometers apart. The networks in an internetwork are connected to each other via specialized devices called *routers* or *gateways*. The Internet is the ubiquitous internetwork of tens of thousands of networks throughout the world. Figure 11.1 (b) shows an internetwork of four networks. The four networks, **Net1–Net4**, are connected via five routers, **R1–R5**. Not all of the networks are directly connected, and two networks can be connected to each other via multiple routers.

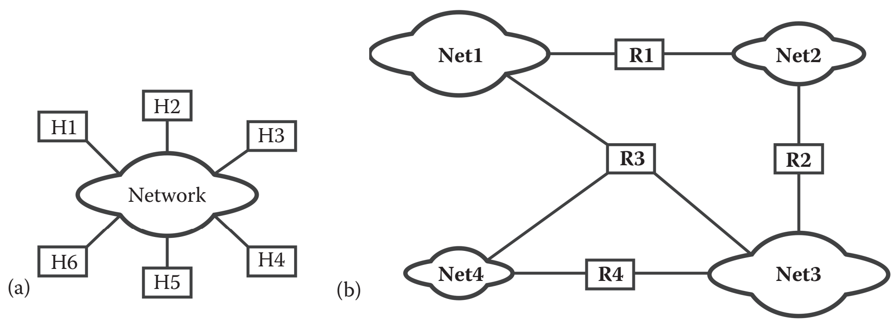


FIGURE 11.1 (a) A network of six hosts; (b) an internetwork of four networks.

In Figure 11.1 (b), for example, **Net2** and **Net4** are not directly connected and **Net3** and **Net4** are connected to each other directly via two routers, **R4** and **R5**. Note that the router **R4** also connects directly **Net3** and **Net1**. Routers such as **R4** that can connect more than two networks are known as *multiport* routers.

11.3 REASONS FOR COMPUTER NETWORKS AND INTERNETWORKS

There are numerous reasons for using networks of computers as opposed to stand-alone personal computers, powerful minicomputers, mainframe computers, or supercomputers. The main reasons include the following:

- Sharing of computing resources: Users of a computer network can share hardware resources including computers, printers, plotters, and scanners, and software resources such as files (data and software).
- Network as a communication medium: A network is an inexpensive, fast, and reliable communication medium between people who live far from each other.
- Cost efficiency: For the same price, you get more computing power with a network of workstations than with a minicomputer or mainframe computer.
- Less performance degradation: With a single powerful minicomputer, mainframe computer, or supercomputer, the work comes to a screeching halt if anything goes wrong with the computer, such as a bit in the main memory going bad. With a network of computers, if one computer crashes, the remaining computers on the network are still up and running, allowing continuation of work.

11.4 NETWORK MODELS

Various questions arose in the design and implementation of networks, and these questions dictated the design of the two best-known network models:

1. The type of physical communication medium, or communication channel, used to connect hardware resources: It can be a simple RS-232 cable, telephone line, coaxial cable, fiber-optic cable, microwave link, or satellite link.

2. The topology of the network—that is, the physical arrangement of hosts on a network: Some commonly used topologies are bus, ring, mesh, and general graph.
3. The set of rules, called protocols, used to allow a host on a network to access the physical medium before initiating data transmission.
4. The protocols used for routing application data (e.g., a Web page) from one host to another in a LAN or from a host in one network to a host in another network in an internetwork.
5. The protocols used for transportation of data from a process on a host to a process on another host in a LAN or from a process on a host in one network to a process on a host in another network in an internetwork.
6. The protocols used by network-based software to provide specific applications such as ftp.

The two best-known network models are the International Standards Organization's Open System Interconnect Reference Model (commonly known as the OSI Seven-Layer Reference Model) and the TCP/IP Five-Layer Model. The OSI model was proposed in 1981 and the TCP/IP model in the late 1970s. In March 1982, the US Department of Defense adopted the TCP/IP model as the standard for all military networks. The TCP/IP model, which has its roots in ARPA, is the basis of the Internet and is, therefore, also known as the Internet Protocol Model. This model consists of five layers, each having a specific purpose and a set of protocols associated with it. The diagram in Figure 11.2 shows the two models, along with an approximate mapping between the two.

Because the TCP/IP model is used in the Internet, this will be our focus. In terms of the six issues previously listed, the first layer in the TCP/IP model deals with the first two issues, the second layer deals with the third issue, the third layer deals with the fourth issue, the fourth layer deals with the fifth issue, and the fifth layer deals with the sixth issue. In terms of their implementation, the first four layers deal with the details of communication between hosts, and the fifth layer deals with the details of the Internet services provided by

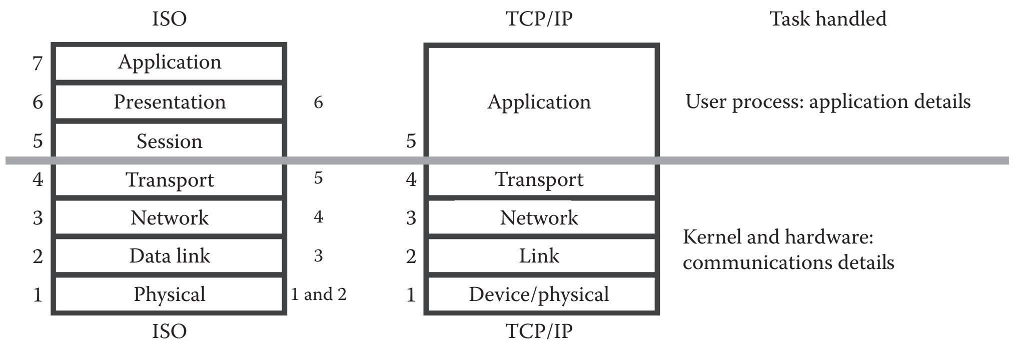


FIGURE 11.2 ISO and TCP/IP layered models, mapping between the two, and the general purpose of a group of layers.

various applications. Most of the first layer is handled by hardware (type of communication medium used, attachments of hosts to the medium, etc.). The *network interface card* (NIC) in a host handles the rest of the first layer and the second layer. Layer 2 consists of medium access control (MAC) addresses, network cards, drivers, and switches. Layers 3 and 4 are fully implemented in the operating system kernel on most existing systems. The first two layers are network hardware specific, whereas the remaining layers work independently of the physical network. On newer gigabit *Ethernet* interfaces where the processing overhead of the network stack becomes significant, the TCP offload engine (TOE) technology is used in the NIC to offload processing of the TCP/IP stack to the network controller.

EXERCISE 11.1

Ask your system administrator: How many hosts are connected on your LAN? What type of computers are they (PCs or workstations)?

EXERCISE 11.2

What is the physical medium for your network (coaxial cable, twisted pair, or glass fiber)? Ask your instructor or system administrator about the topology of your network (bus, ring, etc.).

EXERCISE 11.3

Ethernet is the most commonly used link-level protocol for LANs. Does your LAN use Ethernet? If not, what does it use?

11.5 THE TCP/IP SUITE

Several protocols are associated with various layers in the TCP/IP model. These protocols result in what is commonly known as the *TCP/IP suite*, which is illustrated in Figure 11.3.

The description of most of the protocols in the suite is beyond the scope of this textbook, but we briefly describe the purpose of those that are most relevant to our discussion. As a user, you see the application layer in the form of applications and utilities that can be executed to invoke various Internet services. Some of the commonly used applications are for electronic mail, Web browsing, file transfer, and remote login. We discuss some of the most useful applications in Section 11.8.

11.5.1 TCP and UDP

The purpose of the transport layer is to transport application data from your machine to a remote machine and vice versa. This delivery service can be a simple, best-effort service that does not guarantee reliable delivery of the application data or one that guarantees reliable and in-sequence delivery of the application data. The *User Datagram Protocol* (UDP) provides a best-effort delivery service and the *Transmission Control Protocol* (TCP) offers completely reliable, in-sequence delivery. The UDP is a connectionless protocol; that is, it simply sends the application data to the destination without establishing a virtual

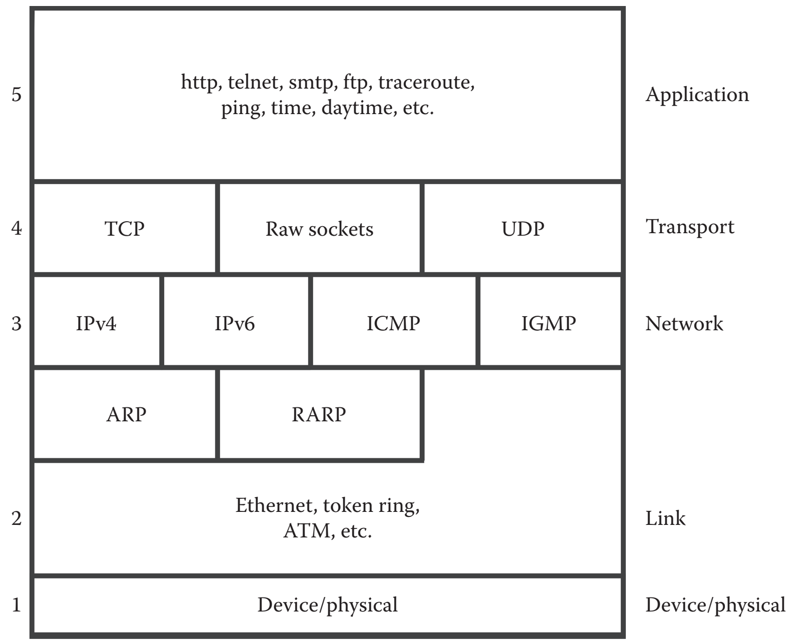


FIGURE 11.3 The TCP/IP.

connection with the destination before transmitting the data. Hence, the UDP software on the sender host does not "talk" to the UPD software on the receiver host before sending data. The TCP is a connection-oriented protocol that establishes a virtual connection between the sender and receiver hosts before transmitting application data, leading to reliable, error-free, and in-sequence delivery of data. Of course, the overhead for establishing the connection makes TCP more costly than UDP. Many well-known Internet applications such as ftp use TCP. Applications where efficiency of data delivery is more important than error-free, in-sequence delivery, such as video streaming, use UDP. In Internet jargon, a data packet transported by TCP is called a *segment* and a data packet transported by UDP is called a *datagram*.

Because multiple client and server processes might be using TCP and/or UDP at any one time, these protocols identify every process running on a host by 16-bit positive integers (0–65,535) called *port numbers*. Port numbers from 0 to 1023 are called *well-known ports* and are controlled by the Internet-Assigned Numbers Authority (IANA). Well-known services such as http are assigned ports that fall in the well-known range (excluding 0). Most of these services allow the use of either TCP or UDP, and the IANA tries to assign the same port number to a given service for both TCP and UDP. For example, the ssh service is assigned port number 22 and the http (Web) server is assigned port number 80, for both TCP and UDP. Most clients can run on any port and are assigned a port by the operating system at the time the client process starts execution. Some well-known clients such as rlogin and ssh require the use of a reserved port as part of the client–server authentication protocol. These clients are assigned ports in the range 513–1023.

11.5.2 Routing of Application Data: The Internet Protocol (IP)

As we mentioned before, the network layer is responsible for routing application data to the destination host. The protocol responsible for this is the *Internet Protocol* (IP), which transports TCP segments or UDP datagrams containing application data in its own packets called *IP datagrams*. The routing algorithm is connectionless, which means that IP routing is best-effort routing and does not guarantee delivery of TCP segments and UDP datagrams. Applications that need guaranteed delivery use TCP as their transport-level protocol or have it built into the application itself. There are two versions of IP: the older version is IPv4 and the new version is IPv6 (commonly known as IPng or Internet Protocol: The Next Generation). In this textbook, we primarily discuss IPv4. The discussion on the actual routing algorithms used by IP is beyond our scope here. However, we describe a key component of routing on the Internet—the IP addressing (naming) scheme to uniquely identify a host on the Internet.

The key to routing is the IP assignment of a unique identification to every host on the Internet. IP does so by uniquely identifying the network the host is on and then uniquely identifying the host on that network. The ID, a 32-bit positive integer in IPv4 and a 128-bit positive integer in IPv6, is known as the host's *IP address*. Every IP datagram has the sender's and the receiver's IP address in it. The sender's IP address allows the receiver to identify and respond to the sender. Hosts and routers perform routing by examining the destination IP address on an IP datagram.

In IPv4, the IP address is divided into three fields: address class, network ID, and host ID. The address class field identifies the class of the address and dictates the number of bits used in the network ID and host ID fields. This scheme results in five address classes: A, B, C, D, and E, with classes A, B, and C being the most common. Figure 11.4

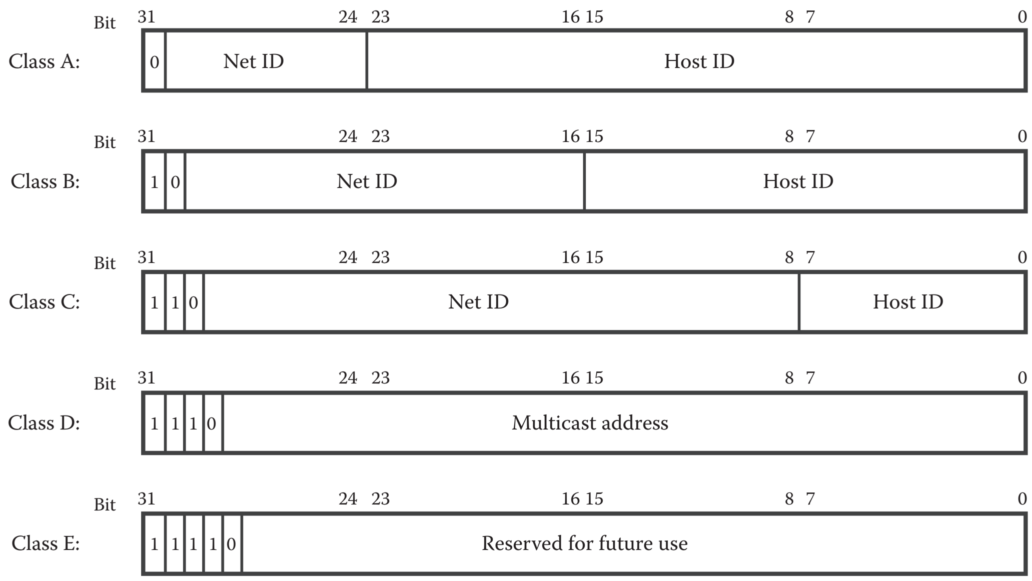


FIGURE 11.4 IPv4 address classes.

shows the structures of the five address types. The IP addresses belonging to classes D and E have special use, and their discussion is beyond the scope of this textbook. A central authority, the Network Information Center (NIC; www.internic.net), assigns all IP addresses.

The maximum number of networks of classes A, B, and C that can be connected to the Internet is given by the expression: $2^7 + 2^{14} + 2^{21}$. Here, 7, 14, and 21 are the number of bits used to specify network IDs in class A, B, and C addresses, respectively. Thus, there are 2^7 class A networks, 2^{14} class B networks, and 2^{21} class C networks. The sum of these numbers gives a total of 2,113,664 networks. Similarly, the number of bits used to identify host IDs in the three classes of addresses can be used to get the maximum number of hosts that can be connected to the Internet. Thus, there are roughly 2^{24} hosts per class A network, 2^{16} hosts per class B network, and 2^8 hosts per class C network. The sum of all the hosts on the three types of networks is a total of 3,758,096,400 hosts. The actual numbers of class A, B, and C networks and hosts are somewhat smaller than numbers shown, due to some special addresses (e.g., *broadcast* and *localhost* addresses). The broadcast addresses are used to address all hosts on a network. The localhost address is used by a host to send a datagram to itself. Hence, an IP datagram with **localhost** as its destination address is never put on the network.

In order to slow down the use of IPv4 addresses and to reduce the growth of routing tables on Internet routers, the Internet Engineering Task Force (IETF; www.ietf.org) introduced Classless Internet-Domain Routing (CIDR) in 1993. Under CIDR, network address spaces in IPv4 are allocated on any address bit boundary, not necessarily on 8-bit sections. There are a lot of classless networks on the Internet.

The number of class A addresses is very small, so these addresses are assigned only to very large commercial organizations, educational institutions, and government agencies, such as US national laboratories, the Massachusetts Institute of Technology (MIT), the University of California at Berkeley, Bell Labs, and NASA. The number of class B addresses is relatively large, and these addresses are assigned to large commercial organizations and educational institutions. Hence, corporations such as IBM and Oracle, educational institutions such as Iowa State University, and numerous other national and international universities have been assigned class B addresses. The total number of class C addresses is quite large, so these addresses are assigned to individuals and small- to medium-sized organizations, such as local Internet service providers, small consulting and software companies, community colleges, and universities.

Although the IPv4 addressing scheme can be used to identify a large number of networks and hosts, it has not been able to cope with the rapid growth of the Internet. Among the many advantages of IPv6 is that an extremely large number of hosts can be connected. With the 128-bit address, the maximum number of hosts on the Internet will increase to roughly 2^{128}, which is greater than 3.4×10^{38}. This number is roughly 6×2^{28} times the present world population. One disadvantage of IPing is that, as the address size is very large, remembering IPv6 addresses becomes very difficult. However, because most users prefer to use symbolic names, remembering IPv6 addresses should not present a problem. Also, some compact notations similar to DDN have been proposed for IPv6 addresses as well. Many large companies and academic institutions have been assigned IPv6 addresses, including Google, IBM, Intel, Microsoft, MIT, UC Berkeley, and Iowa State University. You can use the `host` command to find out if an organization has been assigned an IPv6 address, as in

```
% host berkeley.edu
berkeley.edu has address 169.229.216.200
berkeley.edu has IPv6 address 2607:f140:0:81::f
berkeley.edu mail is handled by 10 mx.berkeley.edu.
%
```

Note that IPv6 addresses are displayed in the *colon-hex notation*. In this notation, two hexadecimal digits are used to specify every byte of the address. A colon (:) is inserted between every two bytes—that is, four hex digits—of an address. If a number between two consecutive colons is not four digits, it represents the least significant nibbles in a two-byte sequence. For example, 81 in the example address in fact is 0081, 0 is 0000, and f is 000f. The right-most hex number for the two-byte sequence is for the least significant two bytes of the 128-bit address. Finally, two consecutive colons between the least significant hex number and previous hex number represent all zeros. The 128-bit IPv6 address of Berkeley.com, consisting of 32 nibbles, is therefore represented as follows:

```
MSD                                                                    LSD
2   6 0 7 F 1 4 0 0 0 0 0 0 0 8 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0  0 F
```

Most UNIX commands and tools have been enhanced to handle IPv6 addresses in the colon-hex notation, as in

```
% host 2610:130:101:100::9
9.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.1.0.1.0.1.0.0.3.1.0.0.1.6.2.i
p6.arpa domain name pointer thumb.iastate.edu.
%
```

Here, the hex numbers are shown in the reverse order of significance—that is, least significant hex digit first. As you can see, the IPv6 address is for thumb.iastate.edu. A more detailed discussion on IPv6 addresses, including the purpose of each byte of the address, is beyond the scope of this book.

11.5.2.1 IPv4 Addresses in Dotted Decimal Notation

Although hosts and routers process IPv4 addresses as 32-bit binary numbers, they are difficult for people to remember. For this reason, the IPv4 addresses are given in *dotted decimal notation* (DDN). In this notation, all four bytes of an IPv4 address are written in their decimal equivalents and are separated by dots. Thus, the 32-bit IP address

```
11000000 01100110 00001010 00010101
```

is written as

```
192.102.10.21
```

in DDN. The ranges of valid IP addresses belonging to the five address classes in DDN are shown in Table 11.1. Some of the addresses given in the table are special addresses.

The internetwork shown in Figure 11.5 connects four networks via four routers, **R1–R4**. **Net1** is a class A network, **Net3** is a class B network, and **Net2** and **Net4** are class C networks. The way to identify the class of a network is to look at the left-most decimal number in the IP address of a host on the network—in this case, the IP addresses of the routers. Note that the routers are assigned as many IP addresses as the number of networks they connect. Here, for example, router **R1** connects **Net1** and **Net2** and has IP addresses 121.1.1.1 and 192.102.10.1. Similarly, **R3** is assigned three IP addresses, as it interconnects three networks **Net1**, **Net3**, and **Net4**.

Of the special addresses, 127.0.0.0 (or 127.x.x.x, where x can be any number between 0 and 255), also known as *localhost*, is used by a host to send a data packet to itself. It also is commonly used for testing new applications before they are used on the Internet. Another special address, in which the host ID field is all 1s, is the *directed broadcast address*. This address is used to send a data packet to all hosts on a network—that is, for broadcasting on a local network whose host is using the address as a destination address.

TABLE 11.1 IPv4 Address Classes and Valid IP Addresses

Address	**Range of Valid IP Addresses**	
Class	**Lowest**	**Highest**
A	0.0.0.0	127.255.255.255
B	128.0.0.0	191.255.255.255
C	192.0.0.0	223.255.255.255
D	224.0.0.0	239.255.255.255
E	240.0.0.0	247.255.255.255

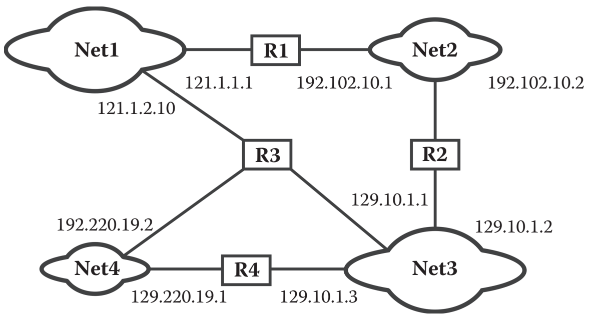


FIGURE 11.5 An internetwork of four networks with one class A, one class B, and two class C networks.

11.5.3 Symbolic Names

People prefer to use symbolic names rather than numeric addresses because names are easier to remember, especially with the transition to the 128-bit-long numeric addresses in IPv6. Also, symbolic names can remain the same even if numeric addresses change. Like its IP address, the symbolic name of a host on the Internet must be unique. The Internet allows the use of symbolic names by using a hierarchical naming scheme. The symbolic names have the format

```
hostname.domain_name
```

where `domain _ name` is the symbolic name referring to the site and is assigned by various registrars whose list is maintained by the NIC. The `hostname` is assigned and controlled by the site that is allocated the `domain _ name`. The `domain _ name` consists of two (or more) strings separated by a period (.). The right-most string in a domain name is called the *top-level domain* (TLD). The string to the left of the right-most period identifies an organization and can be chosen by the organization and assigned to it by the NIC. If the string has already been assigned to another organization under the same top-level domain, another string is assigned in order to keep the domain names unique. There are three types of top-level domains: *special TLDs*, *generic TLDs* (gTLDs), and *country code TLDs* (ccTLDs). According to the IANA's Root Domain Database (http://www.iana.org/domains/root/db), 750 TLDs have been registered at the time of writing. Details of these domains are given in Table 11.2.

For the domain names that consist of more than two strings, the remaining strings are assigned by the organization that owns the domain. Some example domain names are: www.infinione.com, stanford.edu, intel.com, whitehouse.gov, uu.net, omsi.org, cs.berkeley.edu, amazon.com.jp, pucit.edu.pk, www.beaverton.k12.org.us, www.abc.tv, www.nato.int, www.darpa.mil, example.info, and bbc.co.uk. The authorities in a country assign strings to the left of that country's domain. Figure 11.6 illustrates the domain name hierarchy.

Attaching the name of a host to a domain name with a period between them yields the *fully qualified domain name* (FQDN) for the host—for example, cs.stanford.edu, where cs is the name of a host in the Department of Computer Science at Stanford University. However, fully qualified domain names for the hosts on the Internet do not always have three parts. Most organizations allow various groups within the organization to choose the primary names for the hosts that they control and are responsible for. For example, the Department of Computer Science at Stanford, which uses the primary name cs.stanford.edu, uses www.cs.stanford.edu as the FQDN for its HTTP server. The School of Business Administration at Duke, which uses the primary name fuqua.duke.edu, can use the host name www.fuqua.duke.edu for its Web server.

11.5.4 Translating Names to IP Addresses: The Domain Name System

Because Internet software deals with IP addresses and people prefer to use symbolic names, application software translates symbolic names to equivalent IP addresses. This translation involves the use of a service provided by the Internet known as the *Domain*

TABLE 11.2 Top-Level Internet Domains

Domain Type	Top-Level Domain	Assigned to/for
Special	ARPA	Used exclusively for Internet; currently second-e164.arpa, in-addr.arpa, ip6.arpa, uri.arpa, urn.arpa
Generic	ACCOUNTANTS	Knob Town, LLC
	ACTOR	United TLD Holdco, Ltd.
	AERO	Reserved for members of air transport industry
	AIRFORCE	United TLD HoldCo, Ltd.
	ARMY	United TLD HoldCo, Ltd.
	ATTORNEY	United TLD HoldCo, Ltd.
	BEER	Top-Level Domain Holdings Limited
	BIKE	Grand Hollow, LLC
	BIZ	Reserved for businesses
	BUILDERS	Atomic Madison, LLC
	CAREERS	Wild Corner, LLC
	CHRISTMAS	Uniregistry, Corp.
	CHURCH	Holly Fields, LLC
	COM	Reserved for commercial organizations
	COOP	Reserved for cooperative associations
	EDU	Reserved for US postsecondary educational institutions that are accredited by an agency on the US Department of Education's list of Nationally Recognized Accrediting Agencies
	GOV	Reserved for the US government
	INFO	First unrestricted top-level domain since .com, so it can be used by anyone—businesses, marketers, and so on
	INT	Reserved for organizations established by treaties between governments
	MIL	Reserved for the US military
	MUSEUM	Reserved for museums
	NAME	Reserved for individuals
	NET	Intended for internet service providers (ISPs) and telephone service providers
	ORG	Intended for noncommercial communities but all are eligible to register
	PRO	Restricted to credentialed professionals (this domain is being established)
	…	
	ZIP	Charleston Road Registry, Inc.
	ZONE	Outer Falls, LLC
Country Code	AU	Australia
	DE	Germany (Deutschland)
	FI	Finland
	JP	Japan
	PK	Pakistan
	…	
	UK	United Kingdom
	US	United States of America

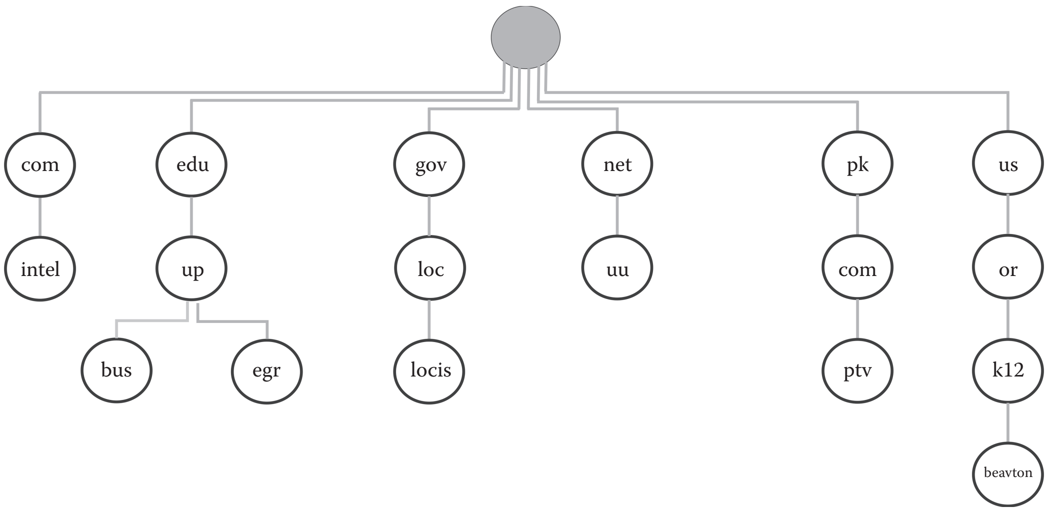


FIGURE 11.6 A portion of the Internet domain name hierarchy.

Name System (DNS). The DNS implements a distributed database of name-to-address mappings. A set of dedicated hosts run server processes called *name servers* that take requests from application software (also called the client software; see Section 11.7) and work together to map domain names to the corresponding IP addresses. Every organization runs at least one name server, often the *Berkeley Internet Name Domain* (BIND) program. The applications use resolver functions such as `gethostbyname` to invoke the DNS service. The `gethostbyname` resolver function maps a hostname (simple or fully qualified) to its IP address, and `gethostbyaddr` maps an IP address to its hostname.

An alternative, and old, scheme for using the DNS service is to use a static hosts file, usually **/etc/hosts**. This file contains the domain names and their IP addresses, one per line. The following command displays a sample **/etc/hosts** file on Solaris.

```
$ more /etc/hosts
#
# Copyright 2009 Sun Microsystems, Inc. All rights reserved.
# Use is subject to license terms.
#
# Internet host table
#
::1             localhost
127.0.0.1       localhost loghost
202.147.169.197 solarissrv
203.128.0.6     yamsrv1.ece.gatech.edu  loghost
203.128.0.1     shahalami
192.168.1.1     suraj-ge0
$
```

There are two problems with this scheme. First, its implementation depends on how the system administrator configures the system. Second, owing to the sheer size of the Internet and its current rate of growth, the static file can be extremely large.

You can use the `ifconfig` command to view and set network interface parameters, including the IP address, localhost, *netmask*, *broadcast address*, and *maximum transmission unit* (MTU). The netmask, also known as the subnetmask, is a bit mask used by TCP/IP to identify whether a host is on a remote network or on a local subnet. A broadcast address is used by the IP layer in a host to send a datagram to all the hosts on a subnet or to a remote network. The Ethernet broadcast address is all 1s. The MTU in a TCP/IP network is the maximum size of an IP datagram (packet) and is dependent on the technology used at the data link layer for connection to other hosts. The following is an example run of the command on Solaris. The command output shows that you are logged on to a host that has localhost and network interfaces each for IPv4 and IPv6 addresses. For IPv4, localhost is 127.0.0.1, with MTU 8232 bytes and netmask ff000000; network IP address is 202.147.169.197 with MTU 1500 bytes and netmask ffffffe0; and broadcast IP address is 202.147.169.223. For IPv6, localhost is ::1 with MTU 8252 bytes and IP address for external traffic is fe80::dad3:85ff:fe7b:21fe with MTU 1500 bytes. The system administrator can enable or disable these interfaces or any of its parameters. Further discussion on the output of the command and its various options for viewing and setting various network parameters is beyond the scope of this book.

```
$ ifconfig -a
lo0: flags=2001000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv4,VIRTUAL>
mtu 8232 index 1 inet 127.0.0.1 netmask ff000000
net0: flags=100001000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4,PHYSR
UNNING> mtu 1500 index 2 inet 202.147.169.197 netmask ffffffe0
broadcast 202.147.169.223
lo0: flags=2002000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv6,VIRTUAL>
mtu 8252 index 1 inet6 ::1/128
net0: flags=120002004841<UP,RUNNING,MULTICAST,DHCP,IPv6,PHYSRUNN
ING> mtu 1500 index 2 inet6 fe80::dad3:85ff:fe7b:21fe/10
$
```

The `ifconfig` command is normally located in the **/sbin** directory on PC-BSD and the **/usr/bin** directory on Solaris. If you get an error message such as `ifconfig`: Command `not found`, you should include the relevant directories in your search path and re-execute the command. You can run the `cat/etc/hosts` command to display the domain names and IP addresses of the hosts on your network.

You can use the `host` command to do the DNS lookup for a host whose domain name is passed as a command line argument to it. The command allows you to display IP address(es) for a domain name or vice versa. In the following session, we use the `host` command to display the IP addresses of the hosts iastate.edu, berkeley.edu, and facebook.com, and the domain names corresponding to two IPv6 addresses. The output of the `host iastate.edu` command displays the IPv4 and IPv6 addresses of iastate.edu (Iowa State University). It also displays that 10 machines handle e-mail at Iowa State. The output of the second command shows that the given IPv6 address is for thumb.iastate.edu. The outputs of the fourth and sixth commands show the IPv4 and IPv6 addresses for berkeley.edu and facebook.com, as well as the fact that 10 hosts each handle e-mail for them. Finally, the output of the last command shows that the name server failed to map the IP address 198.175.96.33 to any domain name, which does not mean that the domain does not exist.

```
% host iastate.edu
iastate.edu has address 129.186.1.99
iastate.edu has IPv6 address 2610:130:101:100::9
iastate.edu mail is handled by 10 mailin.iastate.edu.
% host 2610:130:101:100::9
9.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.1.0.1.0.1.0.0.3.1.0.0.1.6.2.i
p6.arpa domain name pointer thumb.iastate.edu.
% host thumb.iastate.edu
thumb.iastate.edu has address 129.186.1.99
thumb.iastate.edu has IPv6 address 2610:130:101:100::9
% host berkeley.edu
berkeley.edu has address 169.229.216.200
berkeley.edu has IPv6 address 2607:f140:0:81::f
berkeley.edu mail is handled by 10 mx.berkeley.edu.
% host 2607:f140:0:81::f
f.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.1.8.0.0.0.0.0.0.0.4.1.f.7.0.6.2
.ip6.arpa domain name pointer wf-web-prod-03.IST.Berkeley.EDU.
% host facebook.com
facebook.com has address 173.252.120.6
facebook.com has IPv6 address 2a03:2880:2130:cf05:face:b00c:0:1
facebook.com mail is handled by 10 msgin.vvv.facebook.com.
% host 198.175.96.33
Host 33.96.175.198.in-addr.arpa not found: 3(NXDOMAIN)
%
```

On Solaris and some other UNIX systems, you can use the `nslookup` command to do the DNS lookup. Here is a sample run of the command.

```
$ nslookup stanford.edu
Server:          208.67.222.222
Address:   208.67.222.222#53

Non-authoritative answer:
Name:      stanford.edu
Address: 171.67.215.200

$
```

Similarly, you can use the `dig` command to do the DNS lookup on Solaris, some other UNIX systems, and Linux. PC-BSD does not support `dig`. Here is a sample run of the command on Solaris.

```
$ dig stanford.edu
; <<>> DiG 9.6-ESV-R11 <<>> stanford.edu
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 5939
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 3, ADDITIONAL: 0
;; QUESTION SECTION:
;stanford.edu.              IN      A
;; ANSWER SECTION:
stanford.edu.       1800    IN      A       171.67.215.200
;; AUTHORITY SECTION:
stanford.edu.       172800  IN      NS      Argus.stanford.edu.
stanford.edu.       172800  IN      NS      Atalante.stanford.edu.
stanford.edu.       172800  IN      NS      Avallone.stanford.edu.

;; Query time: 566 msec
;; SERVER: 202.147.169.201#53(202.147.169.201)
;; WHEN: Sat Oct 24 23:46:22 PKT 2015
;; MSG SIZE rcvd: 112
$
```

11.5.5 Requests for Comments (RFCs)

The TCP/IP standards are described in a series of documents, known as *Requests for Comments* (RFCs). RFCs are first published as *Internet Drafts* and are made available to all Internet users for review and feedback by placing them in known RFC repositories. After the review process is complete, a draft can become a standard. But not all RFCs are *Internet Standards* documents; some are for information only and others are experimental.

An RFC citation has the following format:

```
#### Title. Authors (up to three). Issue date. (Format: TXT=size-
     in-bytes, PS=size-in-bytes, PDF=size-in-bytes) (Obsoletes
     xxx) (Obsoleted by RFC####) (Updates RFC####) (Updated by
     RFC####) (Also FYI ####) (Status: ssssss)
```

where #### is a four-digit decimal number; `Format` can be `TXT` (ASCII), `PS` (PostScript), and `PDF` (Portable Document Format); and `Status` can be `UNKNOWN`, `PROPOSED STANDARD`, `DRAFT STANDARD`, `STANDARD`, `INFORMATIONAL`, `EXPERIMENTAL`, or `HISTORIC`. Here is an example citation:

```
1180  TCP/IP tutorial. T.J.
      Socolofsky, C.J. Kale. Jan-
      01-1991. (Format: TXT=65494
      bytes) (Status:
      INFORMATIONAL)
```

You can view and download an RFC by accessing any of the repositories maintained on ftp or websites. The most common method of accessing an RFC is to browse the Web page at www.ietf.org/rfc.html. As of the writing of this chapter, there are 7842 RFCs available in the RFC index maintained on this Web page, the last one being submitted in April 2016 as an informational. If you want to be notified of the announcement of a new RFC, you can subscribe to the following distribution list: http://mailman.rfc-editor.org/mailman/listinfo/rfc-dist.

In order to display the text version of an RFC in your browser, type `http://www.ietf.org/rfc/rfcNNNN.txt` into the location field of your browser, where `NNNN` is the RFC number. So, in order to display the text version of RFC 2020, type `http://www.ietf.org/rfc/rfc2020.txt` into the location field of your browser.

The following in-chapter exercises are designed to enhance your depth of understanding of your own network environment by way of learning the domain names and IP addresses of hosts on your network. You will also use the host command to translate domain names to IP addresses, and vice versa.

EXERCISE 11.4

Give the domain names of some hosts on your LAN. Ask your instructor for help if you need any. How did you obtain your answer?

EXERCISE 11.5

List the IP addresses of the hosts identified in Exercise 11.4 in DDN. What is the class of your network (A, B, C, or classless)? How did you find out?

EXERCISE 11.6

Does your network have an IPv6 address? What is its value? Show the command that you used to obtain your answer to this question.

EXERCISE 11.7

Repeat the shell sessions given in this section demonstrating the host command on your system. Do you get the same results? If not, how do the outputs of your commands differ from those shown in this section?

EXERCISE 11.8

Browse the Web page at www.ietf.org/rfc.html, find the citation for RFC1118, and write it down.

11.6 INTERNET SERVICES AND PROTOCOLS

Most users do not understand the intricacies of the Internet protocols and its architecture—nor do they need to. They access the Internet by using programs that implement the application-level protocols for various Internet services. Some of the most commonly used services and the corresponding protocols are listed in Table 11.3. The services are listed in alphabetic order and not according to their frequency of use. You can display the **/etc/services** file on your host to view the Internet services and their well-known port numbers.

The UNIX operating system has some network-related services that are not necessarily available in other operating systems. They include services for displaying all the users logged on to the hosts in a LAN, remote execution of a command, real-time chat in a network, and remote copy. We discuss utilities for most of these services in Section 11.8.

11.7 THE CLIENT–SERVER SOFTWARE MODEL

Internet services are implemented by using a paradigm in which the software for a service is partitioned into two parts. The part that runs on the host that the user is logged on to is called the *client software*. The part that handles client requests and usually starts running when a host boots up is called the *server software*. On the one hand, the server runs forever, waiting for a client request to come. Upon receipt of a request, it services the client request and waits for another request. On the other hand, a client starts running only when a user runs the program for a service that the client offers. It usually prompts the user for input (command and/or data), transfers the client's request to the server, receives the server's response, and forwards the response to the user. Most clients terminate with some sort of "quit" or "exit" command.

Many of the applications are connection-oriented client–server models, in which the client sends a connection request to the server and the server either accepts or rejects the request. If the server accepts the request, the client and server are connected through a *virtual connection*. From this point on, the client sends user commands to the server as requests. The server process serves client requests and sends responses to the client, which sends them to the user in a particular format. Communication between a client and a corresponding server—and the client's interaction with the user—is dictated by the protocol for the service offered by an application. Figure 11.7 shows an overview of the client–server software model.

Thus, when you run a program, such as Firefox, that allows you to surf the Web, an http client process starts running on your host. By default, most clients display the home page

TABLE 11.3 Popular Internet Services and Corresponding Protocols

Service	Protocol
Electronic mail	SMTP (Simple Mail Transfer Protocol)
File transfer	FTP (File Transfer Protocol)
Remote login	SSH (Secure Shell) and TELNET
Web browsing	HTTP (Hyper Text Transfer Protocol) and HTTPS (HTTP Secure)

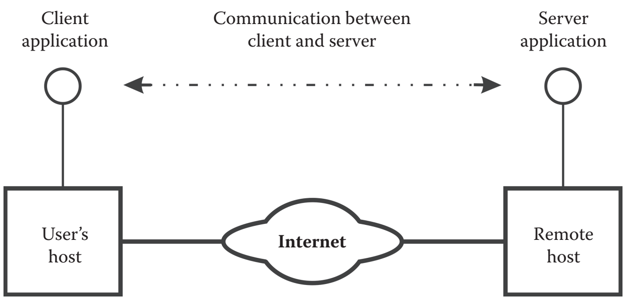


FIGURE 11.7 Depiction of the client–server software model.

of the organization that owns the host on which the client runs, although it can be set to any page, including a blank page. When you want to view the Web page of a site, you give the site's *Uniform Resource Locator* (URL) to the client process. For displaying a home page, the URL has the format:

```
http://host/page
```

where `host` can be the fully qualified domain name or IP address (in DDN) of the computer that has the home page you want to display and `page` is the pathname for the file containing the page to be displayed—for example, http://cnn.com, http://www.stroustrup.com, http://mitadmissions.org/index.php, and http://profiles.stanford.edu/russ-altman. The client tries to establish a connection with the http server process on the site corresponding to the URL. If the site has the http server running and no security protections such as a password are in place, a connection is established between the client and server. The server then sends the Web page to the client, which displays it on the screen, with any audio or video components sent to appropriate devices. Note that `http` can be replaced with `ftp` if you want to access an ftp site through your browser, or with `ssh` if you want to remotely logon via the Secure Shell (SSH) protocol.

You can invoke the client programs for most Internet services by using the corresponding commands, such as `ssh` for the SSH service and `ftp` for the FTP service. Most of these commands permit a domain name or IP address of the host on which the server runs as an argument in the command. Some commands also allow port number as an argument. Client software that has such flexibility built in is known as a *fully parameterized client*. Such clients are important in terms of the flexibility they offer. They also allow testing of updated server software by running it on a port that is not well known and contacting it with the client. A telnet client, discussed in Section 11.8, is a good example of a fully parameterized client.

11.8 APPLICATION SOFTWARE

Numerous programs that implement the application-level protocols just discussed are available on networks of UNIX hosts. Of the most commonly used applications described here, some are available on UNIX- and Linux-based systems only, whereas others are available to all the hosts on the Internet.

11.8.1 Displaying the Host Name

Network-based applications use the **user@host** address format to identify a user on a network on the Internet. You can use the `hostname` and `uname` commands to display the name of the host you are logged on to. On some systems, the host name is shown in the short, simple name format, and on others it is displayed in the long, FQDN format. If you have to identify the host on the network that you are logged on to, you can use the `hostname -s` command to display the short format, which is simply the name of the host (the left-most string in the FQDN format). Some systems do not allow you to use the `-s` option unless you are the superuser. You can use the `uname -n` command to display the host name of the computer that you are logged on to in the FQDN format. The `uname -a` command displays complete information about a host, including the operating system it is running and the name of the CPU. On Solaris, you can use the `–X` option to display the expanded information, one item per line. The following are some examples of the `hostname` and `uname` commands on Solaris.

```
$ hostname
solarissrv
$ uname -n
solarissrv
$ uname -a
SunOS solarissrv 5.11 11.2 i86pc i386 i86pc
$ uname -X
System = SunOS
Node = solarissrv
Release = 5.11
KernelID = 11.2
Machine = i86pc
BusType = <unknown>
Serial = <unknown>
Users = <unknown>
OEM# = 0
Origin# = 1
NumCPU = 4
$
```

The following are the sample runs of hostname and uname on PC-BSD.

```
% hostname
pcbsd-srv
% uname -n
pcbsd-srv
% uname -a
FreeBSD pcbsd-srv 10.0-RELEASE-p6 FreeBSD 10.0-RELEASE-p6 #0
acf484b(releng/10.0): Mon Feb 24 15:14:38 EST 2014  root@avenger:/
usr/obj/root/pcbsd-build-10.0-EDGE/git/freebsd/sys/GENERIC amd64
%
```

11.8.2 Displaying a List of Users Using Hosts on a Network

You can use the `rwho` (remote who) command to display information about the users currently using machines on your network. The output of this command is like that of the `who` command. An output line contains the login name of a user, the computer and terminal the user is logged on to, and the date and time of login. The last field is blank if the user is currently typing at the terminal; otherwise, it shows the number of hours and minutes since the user last typed on the keyboard. You can use the `rwho` command with the `-a` option to include the users who are currently idle. The following session shows a sample output of the command.

```
$ rwho | more
bobk       upibm7:ttyC4      Jul 26 12:03
dfrakes    upibm47:ttyp2     Jul 26 11:49
lulay      upsun17:pts/0     Jul 26 10:17
oster      upsun17:pts/2     Jul 26 12:28
sarwar     upibm7:ttyp2      Jul 26 11:15
$
```

You can use the `rusers` (remote users) command to display the names of the users logged on to the machines on your local network. The login names of the users currently using a machine are displayed one line per machine. The following is a brief description of the command.

SYNTAX

`rusers [options] [host_list]`

Purpose: Display the login names of the users logged on to all the machines on your local network

Output: Information about the users logged on to the hosts on your local network, one line per machine

Commonly used options/features:

- **`-a`** Display a host name even if no user is using it
- **`-l`** Display the user information in a long format similar to that displayed by the who command

The `rusers` command broadcasts a query to all the hosts on the network, asking them to reply with user information. It collects all the replies and displays the information in the order received. The following is a simple run of the command. Note that host names are displayed in a 16-character field, which is why the **edu** part of the names is not completely displayed. As shown in the output of the command, the user **kent** is logged in twice on the **upibm8.egr.up.edu** host, **sarwar** is logged on to **upsun29.egr.up.edu**, and users **kittyt** and **deborahs** are logged on to **upibm6.egr.up.edu**, with **kittyt** logged on twice.

```
$ rusers | more
upibm48.egr.up.e kent kent
```

```
upsun29.egr.up.e sarwar
upibm6.egr.up.ed kittyt kittyt deborahs
$
```

You can display the names of the users logged on to a particular host by specifying the host as an argument to the `rusers` command. The following command displays the login names of the users logged on to the **upibm7** host.

```
$ rusers upibm7
upibm7.egr.up.edu upppp44 upppp upppp26 kathek khnguyen upppp14
leslie sarwar sarwar
$
```

As shown in the following session, you can use the `rusers` command with the `-a` option to display host names even if no user is logged on to them. Doing so allows you to find out the names of all the hosts on your network.

```
$ rusers -a | more
upsun12.egr.up.e
...
upsun27.egr.up.e
upibm48.egr.up.e kent kent
upsun29.egr.up.e sarwar
upsqnt.egr.up.ed
...
upibm3.egr.up.ed
upibm6.egr.up.ed kittyt kittyt deborahs
$
```

11.8.3 Displaying the Status of Hosts on a Network

You can use the `ruptime` (remote uptime) command to display the status of all the computers connected to your LAN. Each line of the output has the following format: computer (host) name, system status (up/down), the amount of time the computer has been up (or down; the number before the + sign indicates the number of days), the number of users logged on to each host, and the load factor for each host. The following is a brief description of the command.

SYNTAX

`ruptime [optins]`

Purpose: Show status of machines on the LAN
Output: Status of machines, including machine name, up/down status, time a machine has been up (or down) for (machine uptime), and the number of users logged on the machine

Commonly used options/features:

- `-a` Include even those users who have been idle for an hour or more
- `-l` Display output after sorting it by load average
- `-t` Display output after sorting it by up time
- `-u` Display output after sorting it by the number of users

The following sessions demonstrate the use of the `ruptime` command with and without the `–u` option.

```
$ ruptime | more
upibm       up   1+09:16,    0 users,   load 0.00, 0.00, 0.00
...
upsun29     up   69+23:51,   2 users,   load 1.48, 1.35, 1.32
$ ruptime -u | more -u
upibm       up   1+09:25,    10 users,  load 0.00, 0.00, 0.00
upibm       up   12+01:01,   4 users,   load 0.10, 0.04, 0.04
upibm       up   8:25,       2 users,   load 0.00, 0.00, 0.03
upibm       up   69+23:57,   1 user,    load 1.30, 1.31, 1.31
...
$
```

Note that **upsun29** has been up for almost 70 days.

In the following in-chapter exercises, you will use the `ruptime`, `rwho`, and `rusers` commands to appreciate their syntax and output. You will also get a feel for what the Internet is primarily used for.

EXERCISE 11.9

Use the `ruptime` command on your system to find out how many hosts are connected to your LAN.

EXERCISE 11.10

Use the `rwho` and `rusers` commands to display information about the users who are currently logged on to the hosts on your network.

EXERCISE 11.11

Ask your friends and fellow students about the two network services they use most often. Which services are they? Which is the more popular of the two? How many people did you ask?

11.8.4 Testing a Network Connection

You can test the status of a network or a particular host on it by using the `ping` command. If the `ping` command does not work on your system, use the type `ping` command to find its location, update your search path, and try the command again. It is normally in the

/usr/sbin directory on Solaris and in the **/sbin** directory on PC-BSD. On PC-BSD, you can also use the whereis `ping command` to find the location of command. The following is a brief description of the command.

SYNTAX

`ping [optins] hostname`

Purpose: Send IP datagrams to **hostname** to test whether it is on the network (or Internet); if the host is alive, it simply echoes the received datagram.
Output: Message(s) indicating whether the machine is alive
Commonly used options/features:

`-c count`	Send and receive count packets
`-f`	Send 100 packets per second or as many as can be handled by the network; only the superuser can use this option
`-s packetsize`	Send **`packetsize`** packets; the default is 56 bytes (plus an 8-byte header)

The following session on PC-BSD illustrates the use of the `ping` command with and without options. The output of the command is different for some systems, with the command displaying the echoed messages until you press `<Ctrl+C>`. We used the `-c` option in the following session to send and receive three messages. The `-c` and `-s` options are used to send and receive three 32-byte messages plus an 8-byte ICMP header. Note that messages greater than 56 bytes are not allowed, as shown by the output of the last command.

```
% ping stanford.edu
PING stanford.edu (171.67.215.200): 56 data bytes
PING stanford.edu (171.67.215.200): 56 data bytes
64 bytes from 171.67.215.200: icmp_seq=0 ttl=231 time=286.052 ms
64 bytes from 171.67.215.200: icmp_seq=1 ttl=231 time=286.336 ms
64 bytes from 171.67.215.200: icmp_seq=2 ttl=231 time=286.038 ms
64 bytes from 171.67.215.200: icmp_seq=3 ttl=231 time=286.395 ms
64 bytes from 171.67.215.200: icmp_seq=4 ttl=231 time=285.864 ms
64 bytes from 171.67.215.200: icmp_seq=5 ttl=231 time=286.047 ms
<Ctrl+C>
--- stanford.edu ping statistics ---
6 packets transmitted, 6 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 285.864/286.122/286.395/0.185 ms
% ping -c 3 stanford.edu
PING stanford.edu (171.67.215.200): 56 data bytes
64 bytes from 171.67.215.200: icmp_seq=0 ttl=231 time=286.161 ms
64 bytes from 171.67.215.200: icmp_seq=1 ttl=231 time=286.175 ms
64 bytes from 171.67.215.200: icmp_seq=2 ttl=231 time=285.770 ms

--- stanford.edu ping statistics ---
3 packets transmitted, 3 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 285.770/286.035/286.175/0.188 ms
```

```
% ping -c 3 -s 32 stanford.edu
PING stanford.edu (171.67.215.200): 32 data bytes
40 bytes from 171.67.215.200: icmp_seq=0 ttl=231 time=285.965 ms
40 bytes from 171.67.215.200: icmp_seq=1 ttl=231 time=285.657 ms
40 bytes from 171.67.215.200: icmp_seq=2 ttl=231 time=307.640 ms

--- stanford.edu ping statistics ---
3 packets transmitted, 3 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 285.657/293.087/307.640/10.291 ms
% ping -c 3 -s 128 stanford.edu
ping: packet size too large: 128 > 56: Operation not permitted
%
```

The Solaris version of the `ping` command simply informs you if the specified host is alive or not. Only the superuser is allowed to use the command with options. Here are a few sample uses. Note that whereas you can use the `–c` option, you are not allowed to use the `–s` option with packet sizes smaller or larger than the default size (56 bytes).

```
$ ping stanford.edu
stanford.edu is alive
$ ping -c 3 stanford.edu
stanford.edu is alive
$ ping -c 3 -s 32 stanford.edu
ping: bad data size: stanford.edu
$ ping -c 3 -s 128 stanford.edu
ping: bad data size: stanford.edu
$
```

You can use the IP address of a host in place of its hostname. For example, you can use `ping 171.67.215.200` instead of `ping stanford.edu`, as in

```
$ ping 171.67.215.200
171.67.215.200 is alive
$
```

11.8.5 Displaying Information about Users

You can use the finger command to display information about users on a local or remote host. The information displayed is extracted from a user's **~/.plan** and **~/.project** files. The following is a brief description of the command.

SYNTAX

```
finger [options] [user-list]
```

Purpose: Display information about the users in **user_list**; without a **user_list**, the command displays a short status report about all the users currently logged on to the specified hosts

Output: User information extracted from the **~/.project** and **~/.plan** files
Commonly used options/features:
`-m` Match **user_list** to login names only
`-s` Display output in a short format

The following session shows the simplest use of the command in which information about a user, **Birch Tree**, on the host is displayed.

```
% finger Birch
Login: tree                              Name: Birch Tree
Directory: /usr/home/tree                Shell: /bin/csh
On since Sat Oct 18 21:55 (PKT) on pts/1 from 102.102.241.10
No Mail.
Plan: To become an informed and efficient UNIX user.
%
```

You can use the finger command with the -s option to display the command's output in a short format and the -m option to match **user-list** to login names only. The finger -m Tree command displays the same information as the finger Birch command if the login name of the user is **tree** (uppercase and lowercase letters are considered the same by the networking commands). However, if the login name of the user is **btree** and the login name **birch** does not exist in the system, the finger command displays the message informing you accordingly.

When run without any argument, the finger command returns the status of all the users who are currently logged on to your machine. The amount of information displayed varies somewhat, depending on the UNIX system that your host runs. The following command runs are on PC-BSD and Solaris systems, respectively.

```
% finger
Login       Name            TTY    Idle  Login  Time    Office Phone
davis       James Davis     pts/0    1d  Aug 14 09:45
sarwar      Mansoor Sarwar  pts/1        Tue    22:43
%

$ finger
Login       Name            TTY       Idle    When    Where
root     Super-User         console  7:09 Sat 16:39
sarwar   Mansoor Sarwar     pts/1         Wed 03:50  39.59.92.15
$
```

You can use the finger command to display information about a user on a host on the Internet, provided the host offers the finger service and has the finger server (fingerd; the finger daemon) running. Remote finger is disabled on most systems. The following command can be used to display information about the user **Pohm** at the **iastate.edu** host.

```
% finger Pohm@iastate.edu
[iastate.edu]
Trying 129.186.1.99...
Iowa State University site-wide finger server.
Use 'finger "/h"@iastate.edu' for help information.

NetID:   Real Name:                Address:                 Phone:
-------- -- ------------------------ ------------------------ --------
avpohm   Pohm, Arthur V
%
```

The format of the output is generally the same, but it can vary from one site to another. Try the `finger davis@iastate.edu` command to see differences in output.

The following session shows the output of the `finger` command when a site contains more than one user with the specified name (**Cohen** in this case) or the specified user does not exist.

```
% finger Cohen@iastate.edu
[iastate.edu]
Trying 129.186.1.99...
Iowa State University site-wide finger server.
Use 'finger "/h"@iastate.edu' for help information.

3 entries found.

NetID:   Real Name:                     Address:
Phone:
--------   ----------------------------- --------------------------
cacohen  Cohen, Cassidy A               3312 West St,   Ames
IA  +1 847 231 6283
ccohen   Cohen, Chantel                 2585 Campus Postal
Sta,  +1 931 494 7748
dcohen   Cohen-Corticchiato, Denis [GE  253 Science I, Ames
IA  +1 515 294 4477
[sarwar@pcbsd-srv] ~%
% finger sarwar@iastate.edu
[iastate.edu]
Trying 129.186.1.99...
Iowa State University site-wide finger server.
Use 'finger "/h"@iastate.edu' for help information.

NOT FOUND: sarwar
%
```

If a host does not run the finger server, the `finger` command displays the `Connection refused` message for you, as in

```
% finger crenshaw@up.edu
[up.edu]
```

```
finger: connect: Connection refused
%
```

If DNS cannot find a mapping for a domain name, the `finger` command returns an appropriate error message. When this happens, you can run the `host` command to find the IP address for the destination host and rerun the `finger` command by using the IP address instead of the domain name.

With `*@hostname` as its argument, the command displays the status of all the users who are currently logged on to **hostname**. Some sites put restrictions on the use of the wild card `*`—for example, requiring the use of at least two characters in all queries. Most sites today do not allow the use of the wild card `*`.

In the following in-chapter exercises, you will use the `ping` and `finger` commands to understand their syntax and various characteristics.

EXERCISES 11.12

Run the `ping` command to determine whether a remote host that you know about is up.

EXERCISES 11.13

Give the command for displaying information about yourself on your UNIX host.

EXERCISES 11.14

Give the command for displaying information about a user on your host, with “John” as his first or last name.

EXERCISES 11.15

Give a command for displaying information about all the users who are currently logged on to the site **cmu.edu**.

11.8.6 Remote Login

Most UNIX systems support three commands that allow you to log on to a remote host. Two are based on the Internet service for remote login, `telnet` and `ssh`, and the other is a BSD command (supported by most UNIX systems) known as `rlogin`. We discuss all three in the following order: `telnet;  rsh` (remote shell), along with all `r` commands (remote insecure commands with commands and data traveling in plaintext); and `ssh` (Secure Shell), along with all `s` commands (remote secure commands with commands and data traveling through encrypted channels).

11.8.6.1 Telnet

The telnet protocol is designed to allow you to connect to a remote computer over a network. This protocol allows you to log on not only to UNIX-based computer systems but also to any computer system that supports the telnet protocol and has a telnet server process running on it. For example, you can connect a UNIX-based computer system to a Windows-based PC. Although you usually need to have a valid account on the remote system, some remote machines allow you to log on without having an account. After the connection has been established, your host or terminal (or display window in a GUI environment) acts as a terminal connected to the remote host. From this point on, every keystroke on your terminal is sent to the remote host. As we mentioned before, telnet is implemented as client–server software. In other words, the host to which you want to connect must have a telnet server process running on a well-known port designated for it, and your command execution starts a telnet client process on your host. The well-known port for the telnet server is port number 23. Because the telnet protocol is based on TCP, the telnet client and server processes establish a virtual connection before prompting you for input. Multiple telnet client processes running on the same host or different hosts can communicate with the same telnet server process—that is, multiple users can use a remote host via telnet.

The UNIX command for starting a telnet client process is `telnet`. The following is a brief description of the command.

SYNTAX

```
telnet [options] [host [port]]
```

Purpose: To connect to a remote system **host** via a network; the host can be specified by its name or IP address in dotted decimal notation

Commonly used options/features:

`-a`	Attempt automatic login
`-l`	Specify a user for login

The telnet client operates in two modes: Input mode and Command mode. When executed without an argument, the client enters Command mode and displays the `telnet>` prompt. When run with a host argument, the client displays the `login:` prompt to take your login name and password. Once a connection has been established between your client and the server, you interact directly with the telnet client. After establishing a connection with the server, the client enters Input mode. In this mode, the client takes character-at-a-time or line-at-a-time Input mode, depending on what the server on the remote host supports; the default mode is character-at-a-time. All input mode data, except `<Ctrl+]>`, known as the telnet escape character, are commands for the remote operating system, transferred to it via the telnet server process. The telnet escape character puts telnet in Command mode. Once it is in Command mode, you can use the `?` or `help` command to display a brief summary of the `telnet` commands. Table 11.4 shows some useful telnet commands and their purpose.

The most common use of the `telnet` command is without an option. The following session shows the use of telnet to log on to another host, **upsun29**, on your network. As shown, after the connection with **upsun29** has been established, you are prompted for your login name and password (shown as a sequence of `*` characters). You must have a valid user account on **upsun29** to be able to use it via telnet.

TABLE 11.4 Commonly Used Telnet Commands

Command	Meaning
`?` or **`help`**	Display a list of telnet commands and their purpose
`close` or **`quit`**	Close the telnet connection
`mode`	Try to enter Line or Character mode
`open host`	Make a telnet connection to **host**
`z`	Suspend the telnet session and return to the local host; resume the telnet session with the `fg` command

```
$ telnet upsun29
Trying 192.102.10.89...
Connected to upsun29.egr.up.edu. Escape character is '^]'.
UNIX(r) System V Release 4.0 (upsun29.egr.up.edu) login: sarwar
Password: **********
Last login: Sat Oct 18 10:05:37 from up You have mail.
DISPLAY = (') TERM = (vt100)
$
```

The following session shows how you can use the `telnet` command for logging on to a host on a remote site (network). Note that you have to use the FQDN of the host to which you want to telnet. The IP address of the host is 191.220.19.2, so you can execute the `telnet 191.220.19.2` command to achieve the same result. The session also shows that you can put telnet into Command mode by pressing `<Ctrl+]>` and run various `telnet` commands before quitting. Some commands, such as `status`, return you to the input mode after completing their task. In the following session, we show that you can use the `z` command to suspend the telnet client and transfer control to the shell on the local host. The `ps` command shows the status of processes on the local machine. The `fg` command reverts to the telnet client. The `<Ctrl+]>` command puts telnet into Command mode, and the `quit` command terminates the telnet session.

```
$ telnet pccaix.sycrci.pcc.edu
Trying 192.220.19.2...
Connected to pccaix.sycrci.pcc.edu. Escape character is '^]'.
telnet (pccaix)
AIX Version 4
(C) Copyrights by IBM and by others 1982, 1996. login: msarwar
msarwar's Password:
Last login: Mon Jul 28 16:57:29 PDT 2003 on /dev/pts/1 from
192.220.11.131
[YOU HAVE NEW MAIL]
$
...
<Ctrl+]>
telnet> z
Suspended
$ ps
```

```
...
$ fg
telnet pccaix.sycrci.pcc.edu
...
<Ctrl+]>
telnet> quit
$
```

As shown in the following session, you can use the `telnet` command to connect to a telnet server that is not running on a well-known port. In this case, the server is running on port number 5000. But be sure to login as **guest** and not register for anything.

```
$ telnet chess.net 5000
Trying 174.129.236.93...
Connected to chess.net.
Escape character is '^]'.
                                 *
WELCOME TO...                   /*\
                                \*/
      /\_/\_/\   _   _   _  *<>*<*>*<>*
     (<>*<>(<>) /*\_/*\_/*\     /*\
    /<>\___\<>/ \_/ \_/ \_/  /\_\*/_/\     _/\_/\_     _/\_/\_
   /<*>/   (<>) (*) <*> (*) (*/_<*>_\*)   /*><*><*\   /*><*><*\
  /<*>/     \/  | \/<*>\/ | \*/*/  *\*/  /<>(  )(<>) /<>(  )(<>)
 (<*>(          <<><>*<><>> <<>(_ _     <<>(___  \/ <<>(___  \/
 (<*>(           \*(<*>)*/  /<><*><>\    (_*_<*>\    (_*_<*>\
  \<*>\     /\   /*/<*>\*\  \<><*><>/        )<*>)       )<*>)
   \<*>\___(<>) <<>(   )<>> <<>(      /\   /<*>)   /\   /<*>)
    \<>/***/<>\ |<>|   |<>| |<>(__/\ (<>)_/<*>/   (<>)_/<*>/   /\
     (<>_<>(<>) (<>)   (<>) (*<>*<>*) \_<>*<>_/    \_<>*<>_/  (<>)
      \/ \/ \/   \/     \/    \/  \/    \/ \/       \/ \/      \/

                        _/\_       /\                 /\
                       (<><>)     (<>)  _/\_/\_  _/\_/**\_/\_
                      / \<>/ \    /**/ (*><*><*) (*><*>**<*><*)
                     /<*>/\(*)\   /<>/  /*/ \(<>)    /<*>/
                    /(*)/  \(*)> /<>/  /*(    \/    /<*>/
                   /(*)/    \(*)/<>/ <>*<>*<>*<>   /<*>/
                  /(*)/      \ /<>/ /*(  /\      /<*>/
                 / \/         \/\/ /<>\_(<>)    /<*>(
                (<>)          (<>) (_<>_<>_/   (_<*>_)
                 \/            \/   \/ \/        \/
Login Screen By Aussie and Isis.
Welcome to Chess.net! Come in and join the fun!
...
login: guest
```

```
You are "guest173". You may use this name to play unrated games.
...
chess% quit
chess.net: Train with grandmasters, play with friends. (TM)
...
$
```

You can run the `telnet` command with a well-known port number as an optional parameter to invoke various Internet services that are connection triggered—that is, services that respond to a connection request from a client. Although not offered at too many hosts anymore, these services include the finger and daytime services offered at the well-known ports 79 and 13, respectively. In the following session, we use the `telnet` command to connect with the finger server at **iastate.edu** and display information about the user **Pohm**.

```
$ % telnet iastate.edu 79
Trying 129.186.1.99...
Connected to iastate.edu.
Escape character is '^]'.
Pohm
Iowa State University site-wide finger server.
Use 'finger "/h"@iastate.edu' for help information.

NetID:   Real Name:                    Address:              Phone:
---------- ------------------------------ ------------------------ ----------
avpohm   Pohm, Arthur V
Connection closed by foreign host.
$
```

11.8.6.2 The `rlogin` Command

The `rlogin` command allows you to log on to a host on your local network. All the hidden files that are executed for a regular login are also executed for remote login. After logging on, therefore, you are put in your home directory and your login shell is executed. As we mentioned before, the `rlogin` command was originally designed for BSD UNIX, but it works on all the systems that have BSD support built into them (e.g., AIX, which is based on System V). The following is a brief description of the `rlogin` command.

SYNTAX

`rlogin [options] host`

Purpose: To connect to a remote UNIX **host** via a network; the host can be specified by its name or IP address in the dotted decimal notation

Commonly used options/features:

`-e C`	Set the escape character to `C` (the default is ~)
`-l user`	Use **`user`** as the login name on the remote host

You can use the `rlogin` command to log on to a UNIX host on your network, provided you have a valid login name and password on the remote host. The following session shows how you can use the command to connect to a UNIX host **upsun29** on your network. Note that, unlike the setup on our system, the `rlogin` command might not prompt you for a `login` name and password if they are the same on the local and remote hosts. Because the rlogin command can be used with an IP address in place of the hostname, the `rlogin 192.102.10.29` command accomplishes the same task. After using the remote system, you can use the `logout` command to log out from the remote system and return to the local system. In the following session, we also show that you can use the `hostname` and `whoami` commands to confirm that you are logged in as the same user (**sarwar**) and that the machine you log on to is **upsun29**.

```
$ rlogin upsun29
Password: xxxxxx
...
You have mail. DISPLAY = (') TERM = (vt100)

$ hostname
upsun29.egr.up.edu
$ whoami
sarwar
$ logout
Terminal session terminated by sarwar rlogin: connection closed.
$ hostname
upsun25.egr.up.edu
$
```

You can use the `-l` option to log on remotely with a login name different from the one you used to log on to the local host. You can use the following command to log on to the remote host **upsun** with the username perform. (Note that **upsun** is on the same LAN as your host.) The `rlogin` command prompts you for your password, and you must have the password for the user for a successful login. If you do not have the right password, `rlogin` lets you try other login names (or the same login name) a few times.

```
$ rlogin upsun -l perform
Password: xxxxxx
Last login: Mon Nov 20 12:08:12 from upsun21.up.edu
SunOS 5.11 11.2 (UPSUN_SERVER) #5: Fri Nov 14 17:31:44 PST 2014
DISPLAY 5 (upx46:0.0)
TERM 5 (vt100)
$ whoami
perform
$ hostname
upsun.egr.up.edu
$
```

The following session shows remote login to a UNIX host that is not on your local network. Here, **cs00.syi.pcc.edu** is the FQDN of a host on the Internet, and the user is **msarwar**. Of course, you must enter the valid password for **msarwar** to be able to log in. You could have used the `rlogin -l msarwar@cs00.syi.pcc.edu` command to establish this session.

```
$ rlogin cs00.syi.pcc.edu -l msarwar
msarwar's Password: xxxxxx
Last login: Wed Nov 22 06:41:41 2003 on /dev/pts/2 from upibm7.
egr.up.edu
[YOU HAVE NEW MAIL]
$ whoami
msarwar
$ hostname
cs00
$
```

In the following in-chapter exercises, you will use the `rlogin` and `telnet` commands to understand how they can be used to log on to a remote host on your network or on the Internet.

EXERCISE 11.16

Use the `telnet` and `rlogin` commands to establish remote login sessions on a host on your LAN. Note that you will not be able to establish these sessions if the remote system does not run the corresponding server processes (e.g., `telnetd` or `in.telnetd`; use the `ps -el` command to determine this).

EXERCISE 11.17

Try to establish a telnet session with the host **locis.loc.gov** and browse through the library at your pace.

EXERCISE 11.18

Repeat all of the sessions shown in this section on telnet on your system. Do your results match ours?

11.8.7 Remote Command Execution

You can use the `rsh` (remote shell) command to execute a command on a remote host on your local network. Remote login is a relatively time-consuming process, but the `rsh` command gives you a faster way to execute commands on remote machines if the purpose of your remote login is to execute only a few commands. We used this command in Chapter 9 to illustrate the power of I/O redirection in UNIX; now we discuss it formally. The following is a brief description of the command.

SYNTAX

```
rsh [options] host [command]
```

Purpose: To execute a command on a remote machine, **host**, on the same network; the `rlogin` command is executed if no `command` is specified

Commonly used options/features:

`-l user` Use **user** as the login name on the remote host

When you execute a command on a remote machine, your current directory on the remote machine is set to your home directory and your login shell on the remote machine is used to execute the command. Only the shell environment hidden files (**.cshrc** for the C shell, **.chrc** for the C shell, etc.) are executed before the `rsh` command is executed. The general environment files (**.login** and **.profile**) are not executed. The standard files (**stdin**, **stdout**, and **stderr**) for the remote command are attached to the standard files used for your local commands—that is, your terminal by default. Thus, when I/O redirection is used, the redirected files are taken from your local machine unless the command to be executed is enclosed in single quotes. We discuss this concept later in this section, and in all cases we assume that **upsun10** is the local machine and that **upsun29** is the remote machine.

The following command line executes the `ps` command on **upsun29**, a *trusted host*, and its output is sent to the display screen of **upsun10** (the local machine). The semantics of the command line are depicted in Figure 11.8.

```
$ rsh upsun29 ps
 PID  TTY    TIME   CMD
6525 pts/0   0:02   -bash
6565 pts/0   0:00   -bash
6566 pts/0   0:00   sort data | uniq > sorted_data
$
```

The following command line executes the `sort students` command on **upsun29**, taking input from the **students** file on **upsun29**, and sends the results back to the **sorted_students** file on the local machine, **upsun10**. If the students file does not exist, the error message is also sent back to **upsun10**. The semantics of the command are illustrated in Figure 11.9.

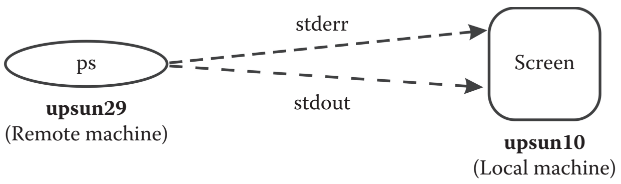


FIGURE 11.8 Semantics of the `rsh upsun29 ps` command.

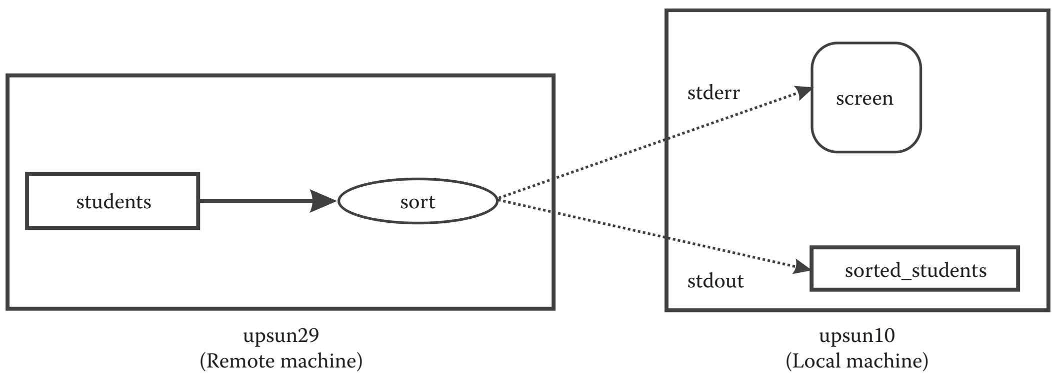


FIGURE 11.9 Semantics of the `rsh upsun29 sort students > sorted_students` command.

```
$ rsh upsun29 sort students > sorted_students
$
```

In the following command line, however, the `sort` command takes input from the **students** file on the local machine, **upsun10**. As with the previous command, the output is sent to the **sorted_students** file on the local machine.

```
$ rsh upsun29 sort < students > sorted_students
$
```

If you want the `sort` command to take input from the local file **students** and store the sorted results in the **sorted_students** file on the remote machine, you must quote the `remote` command with output redirection, as in:

```
$ cat students | rsh upsun29 'sort > sorted_students'
$
```

You can combine the I/O redirection operators with the pipe operator to create powerful command lines that take input from local and remote files, execute commands on local and remote machines, and send the final results to a file on the local or remote machine. We discussed a few such command lines in Chapter 9, which you should revisit at this time.

When used without an argument (which is optional), the `rsh` command reverts to the `rlogin` command. All the rules for the `rlogin` command apply in this case, and all the hidden files that are executed during a normal login are executed. The following session shows the use of the `rsh` command without an argument. In the first example, the command is used to log on to **upsun29** on the same network. In the second example, the `rsh` command is used to log on to a host on a different network on the Internet.

```
$ rsh upsun29
Last login: Mon Oct 20 18:31:17 from upibm7.egr.up.ed
Sun Microsystems Inc.
```

```
SunOS 5.5.1 Generic May 2014
) SPLAY 5 ( TERM 5 (vt100)
$
$ rsh cs00.syi.pcc.edu -l msarwar
msarwar's Password:
Last login: Mon Oct 20 18:56:07 PST 2014 on /dev/pts/2 from
upibm7.egr.up.edu
[YOU HAVE NEW MAIL]
$
```

11.8.8 File Transfer

You can use the `ftp` (File Transfer Protocol) command to transfer files to and from a remote machine on the same network or another network. This command is commonly used to transfer files to and from a remote host on the Internet. The following is a brief description of the command.

SYNTAX

`ftp [options] [host]`

Purpose: To transfer files from or to a remote **host**
Commonly used options/features:
- **`-d`** Enable debugging
- **`-i`** Disable prompting during transfers of multiple files
- **`-v`** Show all remote responses

As we mentioned earlier in the chapter, the File Transfer Protocol is a client–server protocol based on TCP. When you run the `ftp` command, an ftp client process starts running on your host and attempts to establish a connection with the ftp server process running on the remote host. If the ftp server process is not running on the remote host before the client initiates a connection request, the connection is not made and an `Unknown host` error message is displayed by the `ftp` command. A site running an ftp server process is called an *ftp site*. When an `ftp` connection has been established with the remote ftp site, you can run several `ftp` commands for effective use of this utility. However, you must have appropriate access permission to be able to transfer files to the remote site. Table 11.5 presents some useful `ftp` commands.

Most ftp sites require that you have a valid login name and password on that site to be able to transfer files to or from that site. A number of sites allow you to establish ftp sessions with them by using **anonymous** as the login name and **guest** or your full e-mail address as the password. Such sites are said to allow anonymous ftp. In other words, *anonymous ftp* is a method of downloading public files using the ftp protocol. An anonymous log in is usually download only. If you have an account on the computer of the ftp site (i.e., its not an anonymous login), then you are almost certainly allowed to download and upload files.

The following session illustrates the use of the `ftp` utility to do an anonymous ftp with the site ftp.FreeBSD.org and transfer the compressed kernel for release 10.1. In the process,

TABLE 11.5 A Summary of Useful ftp Commands

Command	Meaning
`! [cmd]`	Runs `cmd` on the local machine; without the `cmd` argument, invokes an interactive shell
`Help [cmd]`	Displays a summary of `cmd`; without the `cmd` argument, displays a summary of all ftp commands
`ascii`	Puts the ftp channel into ASCII mode; used for transferring ASCII-type files such as text files
`binary`	Puts the ftp channel into binary mode; used for transferring non-ASCII files such as files containing executable codes or pictures
`cd`	Changes directory; similar to the UNIX `cd` command
`close`	Closes the ftp connection with the remote host, but stays inside ftp
`dir remotedir localfile`	Saves the listing of **remotedir** into **localfile** on the local host; useful for long directory listings, as pipes cannot be used with the ftp commands
`get remotefile [localfile]`	Transfer **remotefile** to **localfile** in the present working directory on the local machine; if **localfile** is not specified, **remotefile** is used as the name of the local file
`ls [dname]`	Shows contents of the designated directory **dname**; current directory if none specified
`mget remotefiles`	Transfers multiple files from the remote host to the local host
`mput localfiles`	Transfers multiple files from the local host to the remote host
`open [hostname]`	Attempts to open a connection with the remote host; prompts if hostname not specified as parameter
`put localfile [remotefile]`	Transfers **localfile** to **remotefile** on the remote host; if **remotefile** is not specified, use **localfile** as name of remote file
`quit`	Terminates the ftp session
`user [login_name]`	If unable to log on, log on as a user on the remote host by specifying the **user_name** as the command argument; prompt appears if **user_name** is not specified

we demonstrate the use of the `ftp` commands `cd`, `get`, `ls`, and `pwd`. We also demonstrate execution of the `ls` command on the local host. Finally, we terminate the ftp session with the `quit` command. This site does not require any password for anonymous ftp. Thus, you hit `<Enter>` when prompted for a password.

```
% ftp ftp.FreeBSD.org
Trying 149.20.53.23:21 …
Connected to ftp.geo.FreeBSD.org.
220 This is ftp0.isc.freebsd.org - hosted at ISC.org
Name (ftp.FreeBSD.org:sarwar): anonymous
331 Please specify the password.
Password: <Enter>
230-
230-This is ftp0.isc.FreeBSD.org, graciously hosted by
230-Internet Systems Consortium - ISC.org.
230-
230-FreeBSD files can be found in the /pub/FreeBSD directory.
```

```
230-
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd pub/FreeBSD
250-ISO images of FreeBSD releases may be found in the releases/
ISO-IMAGES
250-directory. For independent files and tarballs, see individual
250-releases/${machine}/${machine_arch} directories. For example,
250-releases/amd64/amd64 and releases/powerpc/powerpc64.
250 Directory successfully changed.
ftp> pwd
Remote directory: /pub/FreeBSD
ftp> cd releases/i386
250 Directory successfully changed.
ftp> pwd
Remote directory: /pub/FreeBSD/releases/i386
ftp> ls
229 Entering Extended Passive Mode (|||55844|).
150 Here comes the directory listing.
lrwxr-xr-x   1 ftp    ftp   17 Jan 17  2014 10.0-RELEASE ->
i386/10.0-RELEASE
lrwxr-xr-x   1 ftp    ftp   13 Oct 22 15:43 10.1-RC3 ->
i386/10.1-RC3
drwxrwxr-x   13 ftp   ftp   25 Jun 20  2013 8.4-RELEASE
lrwxr-xr-x   1 ftp    ftp   16 Jan 16  2013 9.1-RELEASE ->
i386/9.1-RELEASE
lrwxr-xr-x   1 ftp    ftp   16 Sep 30  2013 9.2-RELEASE ->
i386/9.2-RELEASE
lrwxr-xr-x   1 ftp    ftp   16 Jul 11 14:21 9.3-RELEASE ->
i386/9.3-RELEASE
drwxrwxr-x   3 ftp    ftp   5 Jul 02 16:26 ISO-IMAGES
-rw-rw-r--   1 ftp    ftp   637 Nov 23  2005 README.TXT
drwxrwxr-x   8 ftp    ftp   9 Oct 23 17:21 i386
226 Directory send OK.
ftp> cd 10.1-RC3
250 Directory successfully changed.
ftp> ls
229 Entering Extended Passive Mode (|||64206|).
150 Here comes the directory listing.
-rw-r--r--    1 ftp      ftp          661 Oct 22 01:41 MANIFEST
-rw-r--r--    1 ftp      ftp     62269164 Oct 22 01:41 base.txz
-rw-r--r--    1 ftp      ftp      1428452 Oct 22 01:41 doc.txz
-rw-r--r--    1 ftp      ftp       883080 Oct 22 01:41 games.txz
-rw-r--r--    1 ftp      ftp     77467572 Oct 22 01:41 kernel.txz
-rw-r--r--    1 ftp      ftp     33221748 Oct 22 01:41 ports.txz
-rw-r--r--    1 ftp      ftp     115163384 Oct 22 01:41 src.txz
```

```
226 Directory send OK.
ftp> get kernel.txz
local: kernel.txz remote: kernel.txz
229 Entering Extended Passive Mode (|||61444|).
150 Opening BINARY mode data connection for kernel.txz (77467572
bytes).
100% |**********************************| 75651 KiB 174.60 KiB/s
00:00 ETA
226 Transfer complete.
77467572  bytes received in 07:13 (174.48 KiB/s)
ftp> !ls
kernel.txz
ftp> quit
421 Timeout.
%
```

Once you have established an ftp connection, most sites put you in binary mode so that you can transfer non-ASCII files, such as files containing audio and video clips. You can explicitly put the ftp session into binary mode by using the `binary` command, which ensures proper file transfer. You can revert to ASCII mode by using the `ascii` command.

In the following in-chapter exercise, you will use the `ftp` command to transfer a file from a remote host on the Internet.

EXERCISE 11.19

Establish an anonymous ftp session with the host **ftp.FreeBSD.org** and transfer all the files related to release 10.1 into your system by using the `mget` command.

11.8.9 Remote Copy

You can use the `ftp` command to transfer files to and from a remote host on another network, but doing so requires that you log on to the remote host. You can use the `rcp` (remote copy) command to copy files to and from a remote machine on the same LAN, without logging on to the remote host. This command is not needed in a local area environment if you are using a network-based file system such as the Network File System (NFS). In this case, the storage of your files is completely transparent to you, and you can access them from any host on your network, without specifying the name of the host that contains them. The following is a brief description of the command.

SYNTAX

```
rcp [options] [host:]sfile [host:]dfile
rcp [options] [host:]sfile [host:]dir
```

Purpose: To copy **sfile** to **dfile**

Commonly used options/features:

`-p` Attempt to preserve file modify and access times; without this option, the command uses the current value of `umask` to create file permissions

`-r` Recursively copy files at **sfile** to **dir**

As is obvious from the syntax, you can transfer files from your host to a remote host or from one remote host to another. The `rcp` command fails if the remote host does not "trust" your local host. The name of your local host must be in the **/etc/hosts.equiv** file on the remote machine for it to be a trusted host and for you to be able to use the `rcp`, `rsh`, and `rlogin` commands. You must also have a valid username and password to transfer files to and from the remote host. The format of a line in the file is:

```
hostname [username]
```

A `2` character can precede both a hostname and a username to deny access. A `1` character can be used in place of the hostname or username to match any host or user. The following are a few such entries.

`uphpux sarwar`	Allows access to the user **sarwar** on the host **uphpux**
`+ sarwar`	Allows the user **sarwar** access from any host
`upaix-sarwar`	Denies access to the user **sarwar** on the host **upaix**
`-pc1`	Denies access to all users on the host **pc1**
`pccvm`	Allows access to all users on the host **pccvm**

If the host you are using is not a trusted host—that is, it is not listed in the **/etc/hosts.equiv** file—you need to create an entry in a file called **.rhosts** in your home directory on the remote host that contains the name of the host from which you would use the `rcp` command and your login name on this host. Thus, if the remote host is **upsun29** and you want to use the `rcp` command on a host called **upsun10**, the entry in the **~/.rhosts** file on **upsun29** will be the following (assuming that your login name on **upsun10** is **sarwar**).

```
upsun10     sarwar
```

This entry informs the networking software, which is in the UNIX kernel, that **sarwar** is a trusted user on the host **upsun10**.

The following `rcp` command copies all the files with the **.html** extension from your **~/myweb** directory to the **~/webmirror** directory on **upsun29**.

```
$ rcp ~/myweb/*.html upsun29:webmirror
$
```

The following `rcp` command copies the files **Chapter[1–9].doc** from the **~/unixbook** directory on your system to the **~/unixbook.backup** directory on **upsun29**.

```
$ rcp ~/unixbook/Chapter[1-9].doc upsun29:unixbook.backup
$
```

The following `rcp` command copies all C and C++ source files from your **~/ece446/projects** directory on **upsun29** to your **~/swprojects.backup** directory on your machine.

```
$ rcp upsun29:ece446/projects/*.[c,C] ~/swprojects.backup
$
```

As we mentioned before, you can also use the `rcp` command to copy files from one remote host to another remote host. The following `rcp` command is used to copy the whole home directory from the **www1** host to the **www2** host. Note the use of the `-r` option to copy subdirectories recursively and use of the `-p` option to preserve existing modification times and access permissions.

```
$ rcp -rp www1:* www2:
$
```

The following in-chapter exercise gives you practice using the `rcp` command in your environment.

EXERCISE 11.20

Use an `rcp` command to copy a file from your machine to another machine on your network. What command did you use? What did it do?

11.8.10 Secure Shell and Related Commands

The BSD-originated `r` commands (`rsh`, `rcp`, `rlogin`, etc.) are insecure, as is `telnet`. Secure Shell (SSH) is a modern substitute for them that eliminates the need for the **.rhosts** and **hosts.equiv** files, and can authenticate users by cryptographic key; it also increases the security of other TCP/IP-based applications, such as X Windows *forwarding*, which transparently "tunnels" them via SSH-encrypted connections. The program that handles the secure connection protocol is known as the Secure Shell daemon, sshd.

The secure version of the `rsh` command is called `ssh`. Whereas `rsh`, and the other UNIX `r` commands, communicate with the remote host in *cleartext* (sometimes called *plaintext*), `ssh` uses strong cryptography for transmitting data, including commands, password, and files. Whenever your data is transmitted to your network, ssh automatically encrypts it and automatically decrypts it when the data reaches its intended recipient. The result is *transparent encryption*—that is, the sender and receiver are unaware of the encryption/decryption process.

For these reasons, ssh has become a de facto standard for secure terminal connections within a LAN or the Internet. The encrypted sessions used by ssh, scp, and sftp for example, prevent anyone from making sense of the ongoing communication while *packet sniffing* it. Also, authentication methods used in ssh prevent any kind of *spoofing*, such as IP address spoofing.

SSH Version 2 is the default protocol for both PC-BSD and Solaris. It has three components:

1. SSH Transfer Protocol: Handles server identification, encryption algorithm and key encryption, and manages the channel between client and server

2. SSH Authentication Protocol: Verifies the validity of the login role of user on the client machine

3. SSH Channel Protocol: Handles the encrypted channel and logical connection status for a remote shell session, port forwarding, or X11 forwarding

The following subsections assume that the ssh service is installed and has been started by default, on not only your client local computer, but also on the remote server host machine. We do not provide details of how to obtain or install ssh client or server programs, but we do provide instructions for how to do a preliminary configuration of the sshd daemon and how to start/restart the daemon on both PC-BSD and Solaris.

By default, sshd was enabled and started on both our PC-BSD and Solaris systems, and to use the ssh commands shown in this section, we simply had to let the daemons on both local client machine and remote server machines initially exchange encryption keys, as shown in the sample sessions that follow.

For a more complete description of the `ssh`, `scp`, and `sftp` commands, see the man pages for these commands.

11.8.10.1 Preliminary Configuration of ssh and How to Start/Restart the sshd Daemon

The following subsection gives the preliminary configuration information for ssh on PC-BSD and Solaris, and also shows how to start or restart the daemon process if it has not been started by default on those two systems, or has been stopped for some reason.

1. Configuration of ssh on PC-BSD

   If you do not want to type your password when you use ssh, and you use RSA or DSA keys to authenticate, you can run the `ssh-agent` command before running other applications. X users usually do this through their **.xsession** or **.xinitrc** files. See the man page for `ssh-agent` for details.

   a. Use the `ssh-keygen` command to generate a key pair and have it placed in your **$HOME/.ssh** directory. Your public key is **$HOME/.ssh/id_dsa.pub** or **$HOME/.ssh/id_rsa.pub**, depending on whether you use DSA or RSA, respectively, to authenticate.

   b. Send your public key to the person setting you up as a committer so it can be placed into your login in **/etc/ssh-keys/**. By default, on our PC-BSD and Solaris machines, these keys were generated and stored automatically.

2. Start or restart sshd on PC-BSD

   To determine if sshd is running, or to restart the service, run the following command sequence:

```
% /etc/rc.d/sshd start
sshd already running? (pid=1523)
```

```
% /etc/rc.d/sshd restart
Stopping sshd.
Waiting for PIDS: 1523
Starting sshd.
%
```

An alternative way of starting the ssh daemon is to start an sshd server automatically at boot time by adding the following line to the **/etc/rc.conf** file:

```
sshd_enable="YES"
```

Now start the sshd server by typing the following on the command line:

```
% /etc/rc.d/sshd start
```

1. Configuration of ssh on Solaris

   On the local client machine, examine the files **/etc/ssh_config** and **/etc/ssh/sshd_config**, the client-side and server-side configuration files for ssh, respectively. The lines in these files that are uncommented (i.e., no # as the first character), are the configuration settings for your ssh sessions, both as client and as server, respectively.

2. Starting or restarting sshd on Solaris

   To start sshd on the remote server, if it is not already running, run the following command:

   ```
   $ svcadm enable svc:/network /ssh:default
   ```

   To restart sshd on the remote server, type the following:

   ```
   $ svcadm restart svc:/network/ssh:default
   ```

11.8.10.2 The ssh Command

The `ssh` command allows you to perform basically the same tasks that you can perform with the `telnet`, `rlogin`, and `rsh` commands, but in a more secure manner. You are allowed to log in to the remote host or execute a command on it if the ssh server running on the remote host finds your machine name in the **/etc/host.equiv** or **/etc/shosts.equiv** file and your login name is the same on your machine and the remote machine. You are also allowed to log in to the remote host if your login names are different on the local and remote hosts, and your local machine's name and your login name on it are contained in the **~/.rhosts** or **~/.shosts** file. Additional security features are used by ssh, including public-key cryptography, to authenticate a user and its host before allowing the user to log in to the remote machine. We describe only some of the rather basic features of the `ssh` command that are needed for remote login, remote execution of commands, and file transfer between your client and the server machines. Here is a brief description of the `ssh` command:

SYNTAX

`ssh [-option(s)] [-option argument(s)] [command]`

Purpose: ssh (SSH client) is a program for logging into a remote machine and for executing commands on a remote machine. It is intended to replace rlogin and rsh, and provide secure encrypted communications between two untrusted hosts over an insecure network.

Output: A remote connection over a secure channel to a host on a network or the Internet

Common Options and Features:

`-D [bind_address:]port` Specifies a local "dynamic" application-level port forwarding. This works by allocating a socket to listen to port on the local side, optionally bound to the specified **bind_address**.

`-L [bind_address:]port:host:hostport` Specifies that the given port on the local (client) host is to be forwarded to the given host and port on the remote side.

`-l login_name` Specifies the user to log in as on the remote machine. This also may be specified on a per-host basis in the configuration file.

You can log in to a remote host or execute a command on a remote host with the ssh command just like you do with the rsh command. For example, in the following session, from a PC-BSD client machine, you start an ssh session on a remote host ssh server named **192.168.0.12** using the default username (which in this case is the same username you have on the local client machine), and then log out of the remote host:

```
% ssh 192.168.0.12
The authenticity of host '192.168.0.12 (192.168.0.12)' can't be
established.
RSA key fingerprint is 37:a2:db:ce:97:17:ce:37:26:bf:58:9d:a2:ed:1
c:0f.
No matching host key fingerprint found in DNS.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.0.12' (RSA) to the list of
known hosts.
Password: xxxxxx
Last login: Tue Sep 23 08:51:38 2014
OpenIndiana (powered by illumos)    SunOS 5.11    oi_151a9
November 2013
$
...
$ logout
Connection to 192.168.0.12 closed.
%
```

Notice that since this session is the first ssh connection from the local client to the remote server host, an exchange of authentication keys is necessary. On every subsequent connection, the key exchange is not necessary if the keys have not been changed.

Using the following command, you log in to a host **192.168.0.13** with username **sarwar**, use the `uname -a` command to display the machine information of the server-side machine, log out, and use the `uname -a` command on the client machine to display information of the client-side machine. Note that there is no exchange of authentication keys because this is not the first ssh connection between the client and server machines.

```
% ssh -l sarwar 192.168.0.13
Password:
Last login: Sun Oct 12 17:32:20 2014 from 202.147.169.196
Oracle Corporation      SunOS 5.11      11.2    June 2014
$ uname -a
SunOS solarissrv 5.11 11.2 i86pc i386 i86pc
$ logout
Connection to 202.147.169.197 closed.
% uname -a
FreeBSD pcbsd-srv 10.0-RELEASE-p6 FreeBSD 10.0-RELEASE-p6 #0
acf484b(releng/10.0): Mon Feb 24 15:14:38 EST 2014     root@
avenger:/usr/obj/root/pcbsd-build-10.0-EDGE/git/freebsd/sys/
GENERIC  amd64
%
```

We now discuss remote execution of commands with `ssh`. All of the `rsh` commands discussed in Section 11.8.7 will work if you replace `rsh` with `ssh`. Here are some additional commands. The following command executes the `ps -A | grep d$` command on the remote server host **192.168.0.13** (a Solaris machine) and displays (on the local client machine) the status of all the daemons on that remote host. As shown, you will be prompted for a password because of the default configuration of ssh.

```
% ssh 192.168.0.13 ps -A | grep d$
Password: xxxxxx
    0 ?          0:00 sched
    7 ?          0:03 intrd
   47 ?          0:09 dlmgmtd
  304 ?          0:00 nwamd
  535 ?          4:14 syslogd
   39 ?          0:09 netcfgd
   66 ?          3:20 kcfd
  518 ?          0:00 hald-add
   76 ?          0:38 ipmgmtd
   93 ?          0:00 pfexecd
  478 ?          0:00 in.ndpd
16991 ?          0:00 sshd
```

```
...
%
```

The following command executes the `ps -axl | grep d$` command on the remote server host **192.168.0.14** (a PC-BSD machine) and displays (on the local client machine) the status of all the daemons running on the remote host.

```
$ ssh 192.168.0.14 ps -axl | grep d$
Password for sarwar@pcbsd-srv: xxxxxx
   0    8     0 0 -16 0      0    16 psleep   DL   -   2:56.64 [paged
   0   23     0 0 -16 0      0    16 sdflush  DL   -   0:34.12 [softd
1001 6481 6480 0  20 0 181552 25432 select   I    -   2:00.18 akonad
$
```

In the following session, the ps `-el | grep  d$` command runs on the remote Solaris machine and its output is piped to the `grep  sshd$` command that runs on the local PC-BSD machine. The output of the command line is the status of the ssh daemons running on the remote host.

```
% ssh 122.147.110.13 'ps -e | grep d$' | grep sshd$
Password: xxxx
  512 ?           0:07 sshd
25023 ?           0:00 sshd
25024 ?           0:00 sshd
%
```

11.8.10.3 The scp Command

The `scp` command is the secure version of the `rcp` command. It means that copying takes place under encrypted sessions after proper authentication of the local host and user. All of the commands discussed in Section 11.8.9 will work if `rcp` is replaced with `scp`. The following command copies the **prog4.c** file in your current directory into your **~/courses/cs213/programs/** directory on **upsun29**.

```
$ scp prog4.c upsun29:~/courses/cs213/programs/
Password: xxxxxx
...
$
```

You can use the IP address of **upsun29** (122.147.110.17) in place of its symbolic name, as in:

```
$ scp prog4.c 122.147.110.17:~/courses/cs213/programs/
Password: xxxxxx
...
$
```

The following command recursively copies your entire home directory on the **122.147.110.17** host to your current directory on the local machine. Don't forget to type the period after the space character after ~.

```
% scp -r 122.147.110.17:~ .
Password: xxxxxx
temp                              100%    0     0.0KB/s   00:00
.profile                          100%  568     0.6KB/s   00:00
local.login                       100%  170     0.2KB/s   00:00
welcomesir                        100%  139     0.1KB/s   00:00
known_hosts                       100%  397     0.4KB/s   00:00
scp: /export/home/sarwar/.bash_history: Permission denied
local.cshrc                       100%  166     0.2KB/s   00:00
ps.man                            100%   28KB  27.9KB/s   00:00
bigdata                             3%  327MB  11.2MB/s   12:35 ETA
<Ctrl+C> Killed by signal 2.
%
```

Note that we terminate the command with <CTRL+C>, as can be seen on the last line of the command output.

11.8.10.4 The `sftp` Command

The `sftp` command is the secure version of the `ftp` command. It works just like the `ftp` command except that stronger authentication takes place before file transfer takes place, and transfer takes place in encrypted sessions. The following sessions show sample commands executed in sftp.

Connecting and getting help

```
% sftp 192.168.0.12
Password: xxxxxx
Connected to 202.147.169.197.
sftp> ?
Available commands:
bye                           Quit sftp
cd path                       Change remote directory to 'path'
chgrp grp path                Change group of file 'path' to 'grp'
chmod mode path               Change permissions of file 'path'
                              to 'mode'
chown own path                Change owner of file 'path' to 'own'
df [-hi] [path]               Display statistics for current
                              directory or filesystem containing
                              'path'
exit                          Quit sftp
get [-Ppr] remote [local]     Download file
reget remote [local]          Resume download file
```

```
help                          Display this help text
lcd path                      Change local directory to 'path'
lls [ls-options [path]]       Display local directory listing
lmkdir path                   Create local directory
ln [-s] oldpath newpath       Link remote file (-s for symlink)
lpwd                          Print local working directory
ls [-1afhlnrSt] [path]        Display remote directory listing
lumask umask                  Set local umask to 'umask'
mkdir path                    Create remote directory
progress                      Toggle display of progress meter
put [-Ppr] local [remote]     Upload file
pwd                           Display remote working directory
quit                          Quit sftp
rename oldpath newpath        Rename remote file
rm path                       Delete remote file
rmdir path                    Remove remote directory
symlink oldpath newpath       Symlink remote file
version                       Show SFTP version
!command                      Execute 'command' in local shell
!                             Escape to local shell
?                             Synonym for help
sftp>
```

Listing the local present working directory

```
sftp> lpwd
Local working directory: /usr/home/bob
sftp>
```

Listing the remote present working directory

```
sftp> pwd
Remote working directory: /home/bob
sftp>
```

Listing the files in the remote present working directory

```
sftp> ls
Desktop       passwordlist  sshlogin      sshlogin.txt  test1.c
sftp>
```

List the files in the local present working directory

```
sftp> lls
1stxcbdraw     USBint     bsdsetfile1    qt_progs2     test8.c
1stxcbdraw.c   Videos     bsdsetfile2    qt_progs3     vi.txt
1stxcbdraw.c~  alien      bsdsshman      qt_progs4     xcb_events
...Output truncated
sftp>
```

Uploading a single file named **alien** to the remote host

```
sftp> put alien
Uploading alien to /home/bob/alien
alien                              100%   98     0.1KB/s   00:00
sftp>
```

Uploading multiple files that all end in **.txt** to the remote host

```
sftp> mput *.txt
Uploading emacs.txt to /home/bob/emacs.txt
emacs.txt                           100%   20KB  20.5KB/s   00:00
Uploading fvwm.txt to /home/bob/fvwm.txt
fvwm.txt                            100%  458KB 458.2KB/s   00:01
Uploading vi.txt to /home/bob/vi.txt
vi.txt                              100%   50KB  50.3KB/s   00:00
sftp>
```

Getting (or downloading) single or multiple files from the remote to the local system

```
sftp> get testfile
Fetching /home/bob/testfile to testfile
/home/bob/testfile                 100%  22  0.0KB/s   00.00sftp>
```

Switching from one directory to another directory in local and remote machines
On the remote machine:

```
sftp> cd test
sftp>
```

On the local machine:

```
sftp> lcd Documents
sftp>
```

Creating new directories on local and remote machines
On the remote machine:

```
sftp> mkdir test
sftp>
```

On the local machine:

```
sftp> lmkdir Documents
sftp>
```

Removing a directory or file on the remote machine

```
sftp> rm Report.xls
sftp> rmdir Documents
sftp>
```

Note that to remove/delete any directory from the remote location, the directory must be empty.

Exiting from sftp:

```
sftp> quit
%
```

The following in-chapter exercise gives you practice using the `ssh`, `scp`, and `sftp` commands in your environment.

EXERCISE 11.21

Use each of the `ssh`, `scp`, and `sftp` commands to access a remote host on your LAN, execute a command remotely, and copy a file from your machine to another machine on your network. What command did you use? What are their semantics? Write a short description of how you did this, and what the output from the local and remote systems was.

11.8.10.5 Packaged ssh Applications

In addition to the command line ssh operations shown earlier, which you can perform in a terminal or console window, there are two very powerful and expedient graphical front-end application programs readily available on UNIX and Windows/OS X machines that accomplish the same things that ssh, scp, and sftp do.

The first application is called FileZilla, which allows you to do file and directory transfers over an SSH connection. We will consider its capabilities in Chapter 23 on system administration.

The second application is PuTTY, which as we have shown in Chapter 2, allows you to login and execute commands on a UNIX machine from a Windows machine through an SSH connection.

11.8.11 Interactive Chat

You can use the `talk` command for an interactive chat with a user on your host or on a remote host over a network. The following is a brief description of the command.

SYNTAX

```
talk user [tty]
```

Purpose: To initiate interactive communication with **user** who is logged in on a **tty** terminal

The **user** parameter is the login name of the person if he or she is on your host. If the person you want to talk to is on another host, use **login_name@host** for **user**. The **tty** parameter is needed if the person is logged on to the same host more than once.

When you use the `talk` command to initiate a communication request, the other user is interrupted with a message on his or her screen informing that person of the request.

The other user needs to execute the `talk` command to respond to you. That establishes a communication channel, and both users' display screens are divided into two halves. The upper-half contains the text that you type, and the lower half contains the other user's responses. Both you and the other user can type simultaneously. The `talk` command simply copies the characters that you type at your keyboard on the screen of the other user. The chat session can be terminated when either of you presses `<Ctrl+C>`. If you are using the vim editor and your screen is corrupted during the communication, you can use `<Ctrl+L>` to redraw the screen.

Suppose that user **sarwar** wants to talk to another user, **bob**, and that both are logged on to the same host. The following command from **sarwar** initiates a talk request to **bob**.

```
$ talk bob
```

As soon as **sarwar** hits `<Enter>`, the following message is displayed at the top of **bob**'s screen.

```
[Waiting for your party to respond]
Message from Talk_Daemon@upibm7.egr.up.edu at 13:36 …
talk: connection requested by sarwar@upibm7.egr.up.edu.
talk: respond with: talk sarwar@upibm7.egr.up.edu
```

When **bob** runs the `talk  sarwar` command, both **bob**'s and **sarwar**'s screens are divided in half, with the upper halves containing the message `[Connection established]` and the cursor moved to the top of both screens. Both **bob** and **sarwar** are now ready to talk. If **bob** wants to ignore **sarwar**'s request while using a shell, he can simply press `<Enter>`.

If **bob** is logged in on another host—say, **upsun29—sarwar** needs to run the following command to initiate the talk request.

```
$ talk bob@upsun29
```

If **sarwar** is logged in once on **upibm7** and **bob** wants to communicate with **sarwar**, his response to the preceding request should be `talk  sarwar@upibm7`. If **bob** is logged in on **upsun29** multiple times, the following command from **sarwar** initiates a talk request on terminal **ttyp2** (one of the terminals **bob** is logged on to).

```
$ talk bob@upsun29 ttyp2
```

If you want to block all talk requests because users keep bothering you with too many requests, execute:

```
$ mesg n
$
```

This command works only for your current session. If you want to block all talk and write requests whenever you log on, put this command in your **~/.profile** file on Solaris or

~/.login file on PC-BSD. Doing so simply takes away the write permission on your terminal file in the **/dev** directory for your group and others. Thus, you can accomplish the same by using the `chmod 600` command on your terminal file, as shown in the following session. Notice the change in the permissions of the **/dev/pts/2** file before the after the execution of the `mesg n`, `mesg y`, `chmod 600 /dev/pts/2`, and `chmod 620 /dev/pts/2` commands.

```
% who
david            pts/0        Oct  9 08:17 (:0)
sarwar           pts/1        Oct 25 13:23 (182.185.251.51)
% ls -l /dev/pts/2
crw--w----  1 sarwar  tty  0x98 Oct 25 15:46 /dev/pts/2
% mesg n
% ls -l /dev/pts/2
crw-------  1 sarwar  tty  0x98 Oct 25 15:47 /dev/pts/2
% mesg y
% ls -l /dev/pts/2
crw--w----  1 sarwar  tty  0x84 Oct 25 13:45 /dev/pts/2
% chmod 600 /dev/pts/2
% ls -l /dev/pts/2
crw-------  1 sarwar  tty  0x84 Oct 25 13:45 /dev/pts/2
% chmod 620 /dev/pts/2
% ls -l /dev/pts/2
crw--w----  1 sarwar  tty  0x98 Oct 25 15:51 /dev/pts/2
%
```

Without any argument, the `mesg` command displays the current status.

In the following in-chapter exercises, you will use the `talk` command to establish a chat session with a friend on your network and appreciate the various characteristics of the command.

EXERCISE 11.22

Establish a chat session using the `talk` command with a friend who is currently logged on.

EXERCISE 11.23

Run the last shell session on your system to identify the pathname of your terminal file and verify the effect of the mesg n and mesg y commands on the permissions of your terminal file.

11.8.12 Tracing the Route from One Site to Another

The `traceroute` command uses IP protocol's *time to live* (TTL) field to display the route (the names of the routers in the path) that your e-mail messages, `ssh` commands, and downloaded files from an ftp site can take from your host to the remote host, and vice versa.

It also gives you a feel for the speed of the route. Because this command poses some security threats, most system administrators disable its execution. The security threat stems from the fact that, by displaying a route to a host on the Internet, someone can figure out the internal structure of the network to which the host is connected and the IP addresses of some machines on the network. The following is a simple execution of the command to show its output and demonstrate the inner workings of the Internet a bit more. The following command shows the route from our host to **cs.berkeley.edu** on a PC-BSD machine in Portland, Oregon.

```
% traceroute cs.berkeley.edu
traceroute to cs.berkeley.edu (128.32.244.172), 64 hops max, 52
byte packets
 1  qwestmodem.domain.actdsltmp (192.168.0.1)  0.814 ms  0.803 ms
    0.805 ms
 2  ptld-dsl-gw49.ptld.qwest.net (207.225.84.49)  38.010 ms
    38.741 ms  37.543 ms
 3  ptld-agw1.inet.qwest.net (207.225.86.129)  38.800 ms  37.511 ms
    37.248 ms
 4  svl-edge-20.inet.qwest.net (67.14.12.129)  54.657 ms  54.800 ms
    54.614 ms
 5  65.115.64.166 (65.115.64.166)  63.803 ms  58.525 ms  57.449 ms
 6  dc-oak-core1--tri-isp1-10ge.cenic.net (137.164.47.154)  59.694 ms
    80.097 ms  74.864 ms
 7  dc-oak-agg1--oak-core1-10ge.cenic.net (137.164.47.112)
    64.220 ms  64.496 ms  63.003 ms
 8  137.164.50.31 (137.164.50.31)  65.394 ms  61.800 ms  65.839 ms
 9  t2-3.inr-202-reccev.Berkeley.EDU (128.32.0.39)  63.811 ms
    61.207 ms
    t2-3.inr-201-sut.Berkeley.EDU (128.32.0.37)  60.933 ms
10  evans-eecs-br-10GE.EECS.Berkeley.EDU (128.32.255.54)  85.975 ms
    63.822 ms  64.347 ms
11  * * *
12  * * *
13  * * *
%
```

The default probe datagram (packet) length is 40 bytes, but you can specify a larger length after the destination host name, as in `traceroute cs.berkeley.edu 64`. As shown in the first line of the output of our sample run, the PC-BSD version of `traceroute` uses a 52-byte packet size. A line in the trace contains the times taken by the three 52-byte packets sent by `traceroute` as they go from one router (also known as *hop*) to the next. The output also contains the IP addresses of the various routers on the way from our host to the destination host. A total of 13 hops are traversed by anything that goes from our host to **cs.berkeley.edu**. The Berkeley machine is on a class B network with network ID **128.32.244** and host ID **172**. If `traceroute` does not receive a response within a five-second timeout interval, it prints an asterisk (*) for that probe. Some of the asterisks are unexplainable and may be the result of bugs in the BSD (or other relevant operating system) network code.

The following is the output of the `traceroute` command on Solaris. Note that this version uses 30 hops and 40-byte packets. In this case, the route traversed 11 hops. The one-way travel time for data from our host in Portland, Oregon to **cs.berkeley.edu** in Berkeley, California, in both cases is a little over half a second. Note that the gateways that are 11, 12, and 13 hops away (in the case of the PC-BSD version) and 11 hops away (in the case of the Solaris version) either do not send the ICMP `time exceeded` messages or send them with a TTL too small to reach us.

```
$ traceroute cs.berkeley.edu
traceroute to cs.berkeley.edu (128.32.244.172), 30 hops max, 40
byte packets
 1  qwestmodem.domain.actdsltmp (192.168.0.1)  0.833 ms  0.822 ms
    0.713 ms
 2  ptld (207.225.84.49)  40.396 ms  66.589 ms  38.356 ms
 3  ptld (207.225.86.129)  37.581 ms  37.196 ms  37.710 ms
 4  svl-edge-20.inet.qwest.net (67.14.12.129)  54.609 ms  54.626 ms
    54.853 ms
 5  65.115.64.166 (65.115.64.166)  57.936 ms  58.196 ms  58.836 ms
 6  dc-oak-core1--tri-isp1-10ge.cenic.net (137.164.47.154)
    58.465 ms  60.132 ms  61.275 ms
 7  dc-oak-agg1--oak-core1-10ge.cenic.net (137.164.47.112)
    62.278 ms  59.974 ms  61.523 ms
 8  137.164.50.31 (137.164.50.31)  63.568 ms  62.180 ms  62.294 ms
 9  t2-3.inr-201-sut.Berkeley.EDU (128.32.0.37)  58.946 ms t2-3.
    inr-202-reccev.Berkeley.EDU (128.32.0.39)  60.258 ms t2-3.
    inr-201-sut.Berkeley.EDU (128.32.0.37)  60.120 ms
10  evans-eecs-br-10GE.EECS.Berkeley.EDU (128.32.255.54)  62.424 ms
    59.990 ms  59.202 ms
11  * * *
$
```

11.9 IMPORTANT INTERNET ORGANIZATIONS

Table 11.6 lists the names of some of the important organizations that manage the Internet and formulate plans and policies for its growth.

11.10 WEB RESOURCES

Table 11.7 lists useful websites for network- and Internet-related concepts, organizations, and UNIX commands.

SUMMARY

Computer networking began when the UCLA student Charley Kline sent the first successful message on ARPANET at 10:30 p.m. on October 29, 1969. At that time, ARPANET consisted of four hosts. Today, computing without networking is unthinkable because of

TABLE 11.6 Important Organizations that Manage the Internet and Formulate Plans and Policies for Its Growth

Organization	Purpose
Internet Society (ISOC) `www.isoc.org`	An international, nonprofit organization that was established to encourage and promote the use of the Internet. ISOC is the host for Internet Architecture Board (IAB).
Internet Architecture Board (IAB) `www.iab.org`	A group of people responsible for setting policies and standards for the Internet and the TCP/IP suite.
Internet Engineering Task Force (IETF) `www.ietf.org`	An open group of individuals (network designers, vendors, operators, and researchers) who are responsible for the evolution of the Internet architecture and the Internet's smooth operation. IETF has the responsibility to design and test new technologies for the Internet and the TCP/IP suite. IETF is the technical arm of IAB.
Internet Research Task Force (IRTF) `www.irtf.org`	A group of individuals who are responsible for promoting research that is important for the evolution of the Internet in all relevant areas: protocols, applications, architecture, and technology. IETF is the research arm of IAB.
Internet Assigned Numbers Authority (IANA) `www.iana.org`	Assignment of domain names and protocol port numbers for well-known Internet services, such as ftp.
Internet's Network Information Center (InterNIC) `www.internic.net`	Maintains a list of the currently operating registrars of top-level domains, information about new top-level domains, problem reports about registrars, and information about registered domains.
Internet Corporation for Assigned Names and Numbers (ICANN) `www.icann.org`	ICANN is a technical coordination body whose primary objective is to insure the stability of the Internet's system of assigned names and numbers. Every business that wants to become a registrar with direct access to ICANN-designated top-level domains must be accredited by ICANN for this purpose.

the ubiquitous Internet. Web browsing, file transfer, interactive chat, electronic mail, and remote login are some of the well-known services commonly used by today's computer users. The e-commerce phenomenon has changed the way people do everyday chores and conduct business across the globe. UNIX has a special place in the world of networking in general and internetworking in particular because most of the networking protocols were initially implemented on UNIX platforms. Today, UNIX-based computers run a majority of the server processes that provide most of the Internet services.

The core of internetworking software is based on the TCP/IP protocol suite. This suite includes, among several other protocols, the well-known TCP and IP protocols for transportation and routing of application data. The key to routing in the Internet is 32-bit IP addresses (in IPv4) and 128-bit addresses (in IPv6). The most heavily used Internet services are for Web browsing (and all the services that it offers, such as e-commerce and social networking sites), electronic mail, file transfer, and remote login. Not only do UNIX systems support all the Internet services, but they also have additional utilities to support local network activities.

The topics discussed in this chapter include the general structure of a network and an internetwork, networking models, the TCP/IP suite, IP addresses, the domain name system (DNS), Internet protocols and services, and UNIX utilities for performing networking- and

TABLE 11.7 Web Resources for Network and Internet-Related Policies, Documents, and UNIX Commands

URL	Description
www.openssh.com	The home page for OpenSSH, a free version of ssh.
www.iana.org/domain-names.htm	This page gives detailed information about domain names and domain name services.
www.chiark.greenend.org.uk/~sgtatham/putty/	A home page for PuTTY (a free implementation of telnet and ssh for UNIX and Win32 platform).
www.linuxgazette.com/issue64/dellomodarme.html	An introductory article on the `ssh` command suite: `sftp`, `scp`, and so on
www.isi.edu	The home page for the Information Sciences Institute (ISI) at the University of Southern California (USC). ISI is a useful resource for Internet-related information, such as the history of the Internet, country codes, and protocol port numbers for the well-known Internet services.
www.isoc.org/internet/history/	This page contains many documents on the history of the Internet, including its infrastructure, standards, growth, and future.
www.zakon.org/robert/internet/timeline/	This page has a detailed time line for the history of the Internet, including statistical data for the number of users, hosts, and domains served by the Internet.
en.wikipedia.org/wiki/Internet_protocol_suite	This page contains the history of the TCP/IP protocol suite.
www.ietf.org/rfc.html	This page contains all you need to know about RFCs.
www.ietf.org/download/rfc-index.txt	This page contains the RFC index in text form.
www.iana.org/domains/root/db	This page contains the IANA root zone (i.e., top-level domains) database.
www.internetlivestats.com/total-number-of-websites/	This page maintains a live counter for total number of websites in the world.
http://www.worldwidewebsize.com/	This page maintains the size of the Internet in terms of number of webpages.
news.netcraft.com/archives/2014/04/02/april-2014-web-server-survey.html	This page maintains the latest survey about the number and type of web servers on the Internet, and the growth history of Web servers.
www.datacenterknowledge.com/archives/2009/05/14/whos-got-the-most-web-servers/	This page contains the latest information about who has the most number of Web servers.
en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers	List of TCP and UDP port numbers for all well-known and registered ports.
en.wikipedia.org/wiki/List_of_virtual_communities_with_more_than_1_million_users	This page contains the latest information about the number of registered users of the social websites (virtual communities).
zephoria.com/social-media/top-15-valuable-facebook-statistics/	This page maintains the latest top-20 statistics about Facebook.
www.internetworldstats.com/stats.htm	This page maintains the latest statistics about the worldwide users of the Internet.

(*Continued*)

TABLE 11.7 (CONTINUED) Web Resources for Network and Internet-Related Policies, Documents, and UNIX Commands

URL	Description
`ftp.isc.org/www/survey/reports/2014/01/`	This page contains the latest information about the number of Internet domains.
`www.isc.org/services/survey/`	This page contains the latest information about the number of hosts on the Internet.
`www.factshunt.com/2014/01/world-wide-internet-usage-facts-and.html`	This page contains the latest information about worldwide Internet usage facts.
`http://www.ftp-sites.org/`	This page, although fairly dated, contains a list of known ftp sites.
`https://www.freebsd.org/doc/handbook/mirrors-ftp.html`	This page contains a list of worldwide ftp sites for official sources of FreeBSD.
`ftp://ftp.FreeBSD.org/pub/FreeBSD/`	Anonymous ftp site for official FreeBSD sources that allows a large number of simultaneous connections.
`https://en.wikipedia.org/wiki/ARPANET`	This page describes the history of ARPANET and brief early history of the Internet, e-mail, and ftp.

internetworking-related tasks. These utilities are implemented by using the client–server software model. The utilities discussed in this chapter are `finger` (for finding information about users on a host), `ftp` and `sftp` (for file transfer), `ifconfig`, `host`, and `nslookup` (for translation of domain names to IP addresses and vice cera), `ping` (to find the status of a host), `rcp` and `scp` (to remote copy on a UNIX host), `rlogin` (to remote login on a UNIX host), `rsh` and `ssh` (for logging on to a remote host on a network and remote command execution), `ruptime` (to display information about UNIX hosts on a LAN), `rwho` (to display users who are currently logged on to UNIX hosts in a LAN), `talk` (for interactive chat), `telnet` (for remote login), and `traceroute` (for tracing the route of data from your host to a destination host).

QUESTIONS AND PROBLEMS

1. What are computer networks and why are they important?
2. What is an internetwork? What is the Internet?
3. What are the key protocols that form the main pillars of the Internet? Where were they developed?
4. What is an IPv4 address? What is its size in bits and bytes? What is dotted decimal notation? What is the size of an IPv6 address in bits and bytes?
5. What are the classes of IPv4 addresses? Given an IPv4 address in binary, how can you tell which class the address belongs to? How can you tell the class of the address when it is expressed in the dotted decimal notation?
6. What is the DNS? Name the UNIX command that can be used to translate a host name to its IP address.

7. List two domain names each for sites that are in the following top-level domains: edu, com, gov, int, mil, net, org, autos, beer, biz, careers, cancerresearch, church, museum, au, de, ir, kw, pk, and uk. How did you find them? Do not use examples given in this textbook.

8. Read the **ftp://ftp.isi.edu/in-notes/iana/assignments/port-numbers** file to identify port numbers for the following well-known services: ftp, http, time, daytime, echo, ping, ssh, and quote-of-the-day.

9. What is the timeout period for the finger protocol? How did you get your answer?

10. How many users have **Johnston** as part of their name at the **iastate.edu** domain? How did you find out? Write down your command.

11. Give a command that accomplishes the same task that the following command does.

```
rsh upsun29 sort < students > sorted_students
```

12. Show the semantics of the following command by drawing a diagram similar to the ones shown in Figures 14.8 and 14.9. Assume that the name of the local machine is **upsun10**.

```
cat students | ssh upsun29 sort | ssh upsun21 uniq >
sorted_uniq_students
```

13. Display the **/etc/services** file on your system and list the port numbers for well-known ports for the following services: daytime, time, quote-of-the-day (qotd), echo, smtp, and finger. Did you find all of them? Do the port numbers match those found in Problem 8?

14. Use the `telnet` command to get current time via the daytime service at **mit.edu**. Write down your command.

15. Fetch the files **history.netcount** and **history.hosts** from the directory **nsfnet/statistics** using anonymous ftp from the host **nic.merit.edu**. These files contain the number of domestic and foreign networks and hosts on the NSFNET infrastructure. What is the size of Internet in terms of the number of networks and hosts according to the statistics in these files? Although the statistics are somewhat dated, what is your prediction of its size a year from now? Why? Show your work.

16. You create the following entries in your **~/.rhosts** file on a host on your network.

```
host1   john.doe

host2   mike.brich
```

What are the consequences if **john.doe** and **mike.birch** are users on hosts **host1** and **host2** in your network? Both users belong to your user group.

17. Give a command for displaying simple names of all the hosts on your network.

18. Use the `telnet` command to display information about all the users at **mit.edu** who have **Smith** as part of their name. Show the command that you used to obtain your answer.
19. Give the command for displaying page-by-page information about the users who have **Chen** as part of their name at **mit.edu**. How many such users exist?
20. Describe the semantics of the following command. Clearly state which commands are executed locally and which are executed on the remote host. What is the output of the command?

    ```
    ssh cs00.syi.pcc.edu " ps -el | grep d$ " | grep '\<httpd'$
    | wc -l
    ```

21. Use the `traceroute` command to determine the route from your host to loc.gov. What is the approximate travel time for data from your host to locis.loc.gov? If their site is blocking `traceroute` to either skip this question or to go a website that does `traceroute` for you such as traceroute.org or ping.eu/traceroute.
22. Find a host that offers the quote-of-the-day (`qotd`) service. What is the quote of the day today?
23. What kind of network traffic is generated when an IP datagram is sent to **localhost**.
24. Which of the following domains do not have an IPv6 address: google.com, twitter.com, amazon.com, instagram.com, ibm.com? Show your commands and their outputs.
25. How many machines (hosts) do Google and IBM use to handle its mail service? How did you find out? Show the command(s) that you used to obtain your answer, along with the outputs of these commands.
26. Use the `ifconfig` command to give the following information about the local and network interfaces on your machine for IPv4 and IPv6 addresses: localhost addresses and network IP addresses along with their MTUs.
27. What is the domain name of the host with the following IPv6 address: **2a03:2880:2130:cf05:face:b00c:0:1**? Show the command(s) that you used to obtain your answer, along with the outputs of these commands.
28. What is the current RFC count? What is the last RFC about? What is its category (Standard, proposed standard, etc.)? How did you obtain your answer?
29. Describe the semantics of the following commands. In particular, state clearly which commands in the command line execute on the local machine and which execute on the remote machine.

    a. `cat students | ssh 122.147.110.13 'sort | grep David'`

    b. `cat students | ssh 122.147.110.13 'sort' | grep David`

CHAPTER 12

Introductory Bourne Shell Programming

Objectives

- To introduce the concept of shell programming
- To discuss how shell programs are executed
- To describe the concept and use of shell variables
- To discuss how command line arguments are passed to shell programs
- To explain the concept of command substitution
- To describe some basic coding principles
- To discuss the various control structures for Bourne shell scripting
- To explain the concept of functions in the Bourne shell
- To write and discuss some shell scripts
- To describe how Bourne shell scripts may be debugged
- To cover the commands and primitives

```
*, =, ", ', ', &, <, >, ;, |, \, /, [], :, break, case,
continue, exit, export, env, for, if, ls, read, readonly, set,
sh, shift, test, while, until, unset
```

12.1 INTRODUCTION

The Bourne shell is more than a command interpreter. It has a programming language of its own that can be used to write shell programs for performing various tasks that cannot be performed by any existing command. A shell program, commonly known as a *shell* *script*, consists of shell commands to be executed in a shell, one command at a time, and is stored in an ordinary UNIX file. The shell allows use of a read/write storage place, known as a *shell variable*, for users and programmers to use as a scratch pad for completing a task. The shell also contains program control flow commands (also called *statements*) that allow nonsequential execution of the commands in a shell script and repeated execution of a block of commands—similar to high-level programming languages like C.

12.2 RUNNING A BOURNE SHELL SCRIPT

There are three ways to run a Bourne shell script. The first step for all three methods is to make the script file executable by adding the execute permission to the existing access permissions for the file. You can do so by running the following command, where **script_file** is the name of the file containing the shell script.

```
$ chmod u+x script_file
$
```

Clearly, in this case you make the script executable for yourself only. However, you can set appropriate access permissions for the file if you also want other users to be able to execute it. Once you have made the script file executable, you can type `./script _ file` as a command to execute the shell script, as shown:

```
$ ./script_file
... Output of the script if any ...
$
```

If your search path (the `PATH` variable) includes your current directory (.), you can simply use the `script _ file` command, instead of using the `./script _ file` command. For the rest of this chapter, we assume that your `PATH` variable includes your current directory.

As described in Chapter 10, a child of the current shell process executes the script. Thus, with this method, the script executes properly if you are using the Bourne shell but not if you are using any other shell. If you are currently using some other shell, first execute the `/bin/sh` command to run the Bourne shell and then run the `script _ file` command, as shown in the following example. Here, we assume that your current shell is C shell (with the `%` prompt). After the script has completed its execution, we press `<Ctrl+D>` to terminate the Bourne shell and return to C shell.

```
% /bin/sh
$ ./script_file
... Output of the script if any ...
$ <Ctrl+D>
%
```

The second method of executing a shell script is to run the `/bin/sh` command with the script file as its parameter. Thus, the following command executes the shell script in **script_file**.

```
$ /bin/sh script_file
... Output of the script if any ...
$
```

If your `PATH` variable includes the **/bin** directory, you can simply use the `sh` command, instead of using the `/bin/sh` command.

The third method, which is also the most commonly used method, is to force the current shell to execute a script in the Bourne shell, regardless of your current shell. You can do so by beginning a shell script with the following line.

```
#!/bin/sh
```

When your current shell encounters the string `#!`, it takes the rest of the line as the absolute pathname for the shell to be executed, under which the script in the file is executed. If your current shell is the C shell, you can replace this line with a colon (:), which is known as the *null* command in the Bourne shell. When the C shell reads `:` as the first character, it runs a Bourne shell process that executes the commands in the script. The `:` command returns true. We discuss the return values of commands later in the chapter.

Throughout this chapter, we would use the `chmod u+x script _ file` command to make **script_file** executable by the owner of the file and run the script by using the `./script _ file` command.

12.3 SHELL VARIABLES AND RELATED COMMANDS

A *variable* is a main memory location with a name. It allows you to reference the memory location by using its name instead of its address. The name of a shell variable can comprise digits, letters, and underscores, with the first character being a letter or underscore. Because main memory is read/write storage, you can read a variable's value or assign it a new value. For the Bourne shell, the value of a variable is always a string of characters, even if you store a number in it. There is no theoretical limit on the length of a variable's value.

Shell variables can be one of two types: *shell environment variables* and *user-defined variables*. Environment variables are used to customize the environment in which your shell runs and for proper execution of shell commands. A copy of these variables is passed to every command that executes in the shell as its child. Most of these variables are initialized when the **/etc/profile** file executes as you log on. This file is written by your system administrator to set up a common environment for all the users of the system. You can customize your environment by assigning different values to some or all of these variables, as well as define other variables, in your **~/.profile** startup file, which executes when you log on. Table 12.1 lists most of the environment variables whose values you can change. We described some of these variables in previous chapters.

These shell environment variables are *writable*, and you can assign any values to them. Other shell environment variables are *read only*, which means that you can use (read) the values of these variables but cannot change them. These variables are most useful for processing command line arguments (also known as *positional arguments*) or parameters

TABLE 12.1 Some Important Writable Bourne Shell Environment Variables

Environment Variable	**Purpose of the Variable**
`CDPATH`	Contains the names of the directories that are searched, one by one, by the `cd` command to find the location of the directory passed to it as a parameter; the `cd` command searches the current directory if this variable is not set
`EDITOR`	Contains the name of the default editor used in programs such as an e-mail program
`ENV`	Contains the path along which UNIX looks to find configuration files
`HOME`	Contains the name of the directory where the login shell places you in the directory structure when you first log on
`MAIL`	Contains the name of user's system mailbox file
`MAILCHECK`	Contains a number that specifies how often (in seconds) the shell should check a user's mailbox for new mail and inform the user accordingly
`PATH`	Contains the user's search path—that is, the sequence of directories that a shell searches to find an external command or program
`PPID`	Contains the process ID of the parent process
`PS1`	Contains the primary shell prompt that appears on the command line, usually set to $
`PS2`	Contains the secondary shell prompt displayed on second line of a command if the shell thinks that the command is not finished, typically when the command terminates with a backslash (\\), the escape character. Usually set to >.
`PWD`	Contains the absolute pathname of the current working directory
`TERM`	Contains the type of user's console terminal

TABLE 12.2 Some Important Read-Only Bourne Shell Environment Variables

Environment Variable	**Purpose of the Variable**
`$0`	Name of program
`$1-$9`	Values of command line arguments 1–9
`$*`	Values of all command line arguments
`$@`	Values of all command line arguments; each argument individually quoted if `$@` is enclosed in quotes, as in `"$@"`
`$#`	Total number of command line arguments
`$$`	Process ID (PID) of current process
`$?`	Exit status of most recent command
`$!`	PID of most recent background process

passed to a shell script at the command line. Examples of command line arguments are the source and destination files in the `cp` command. Some other read-only shell variables are used to keep track of the process ID of the current process, the process ID of the most recent background process, and the exit status of the last command. Some important read-only shell environment variables are listed in Table 12.2. These read-only variables are established at the time the process is invoked rather than at the login time, as with other environment variables.

User-defined variables are used within shell scripts as temporary storage places whose values can be changed when the program executes. These variables can be made read only as well as passed to the commands that execute in the shell script in which they are defined. Unlike most other programming languages, in the Bourne shell programming language you do not have to declare and initialize shell variables. An uninitialized shell variable is initialized to a null string by default.

You can display the names of all shell variables (including user-defined variables) and their current values by using the `set` command without any parameters. As described later in this chapter, the `set` command can also be used to change the values of some of the read-only shell environment variables. The following is a sample run of the `set` command on our machine.

```
$ set
BLOCKSIZE=K
CLICOLOR=true
EDITOR=vi
GROUP=faculty
HOME=/home/sarwar
HOST=pcbsd-srv
HOSTTYPE=FreeBSD
IFS='
'
LANG=en_US.UTF-8
LOGNAME=sarwar
MACHTYPE=x86_64
MAIL=/var/mail/sarwar
MANPATH=/usr/share/man:/usr/local/man:/usr/share/openssl/man:
/usr/pbi/man:/usr/local/lib/perl5/5.16/man:/usr/local/lib/
perl5/5.16/perl/man
MORE=-erX
NO_PROXY=127.0.0.1,localhost
OPTIND=1
OSTYPE=FreeBSD
PAGER=more
PATH=/usr/local/share/pcbsd/bin:/sbin:/bin:/usr/sbin:/usr/bin:
/usr/games:/usr/pbi/bin:/usr/local/sbin:/usr/local/bin:/home/
sarwar/bin
PPID=35570
PS1='$ '
PS2='> '
PS4='+ '
PWD=/home/sarwar
REMOTEHOST=static-host202-147-168-98.link.net.pk
SHELL=/bin/csh
SHLVL=1
SSH_CLIENT='202.147.168.98 50256 22'
```

```
SSH_CONNECTION='202.147.168.98 50256 202.147.169.196 22'
SSH_TTY=/dev/pts/1
TERM=xterm-color
USER=sarwar
VENDOR=amd
_=MAC
no_proxy=127.0.0.1,localhost
$
```

You can also use the env (System V) and printenv (BSD) commands to display the names of the environment variables and their values, but the list is not as complete as the one displayed by the set command. In particular, the output does not include any user-defined variables. The following is a sample output of the env command on the same system that we ran the set command on.

```
$ env
VENDOR=amd
SSH_CLIENT=202.147.168.98 50256 22
LOGNAME=sarwar
PAGER=more
LANG=en_US.UTF-8
no_proxy=127.0.0.1,localhost
OSTYPE=FreeBSD
NO_PROXY=127.0.0.1,localhost
MACHTYPE=x86_64
CLICOLOR=true
MAIL=/var/mail/sarwar
PATH=/usr/local/share/pcbsd/bin:/sbin:/bin:/usr/sbin:/usr/bin:
/usr/games:/usr/pbi/bin:/usr/local/sbin:/usr/local/bin:/home/
sarwar/bin
EDITOR=vi
HOST=pcbsd-srv
REMOTEHOST=static-host202-147-168-98.link.net.pk
PWD=/home/sarwar
GROUP=sarwar
TERM=xterm-color
SSH_TTY=/dev/pts/1
HOME=/home/sarwar
USER=sarwar
SSH_CONNECTION=202.147.168.98 50256 202.147.169.196 22
HOSTTYPE=FreeBSD
SHELL=/bin/csh
MORE=-erX
MANPATH=/usr/share/man:/usr/local/man:/usr/share/openssl/man:/usr/
pbi/man:/usr/local/lib/perl5/5.16/man:/usr/local/lib/perl5/5.16/
perl/man
```

```
BLOCKSIZE=K
SHLVL=1
$
```

In the following in-chapter exercises, you will create a simple shell script and make it executable. Also, you will use the `set`, `printev`, and `env` commands to display the names and values of the shell variables on your system.

EXERCISE 12.1

Display the names and values of all the shell variables on your UNIX machine. What command(s) did you use?

EXERCISE 12.2

Create a file that contains a shell script comprising the `date` and `pwd` commands, one on each line. Make the file executable and run the shell script. List all the steps for completing this task.

12.3.1 Reading and Writing Shell Variables

The syntax for the assignment statement in the Bourne shell is as follows. The command syntax `variable=value` comprises what is commonly known as the *assignment statement*, and its purpose is to assign a value to a variable. The evaluation of the assignment statement is right to left. Thus, `value` is evaluated first and then assigned to `variable`. If there is a problem in evaluating `value`, an error is reported.

SYNTAX

`variable1=value1 [variable2=value2 … variableN=valueN]`

Purpose: Assign values **`value1`**, …, **`valueN`** to variables **`variable1`**, …, **`variableN`**, respectively; no space allowed before or after the equal sign

Note that there is no space before and after the equals sign (=) in the syntax. If a value contains spaces, you must enclose the value in quotes. Single and double quotes work differently, as discussed later in this section. You can refer to (i.e., access) the current value of a variable by placing a dollar sign ($) before the variable name; there is no space between `$` and the variable name. You can use the `echo` command to display the values of shell variables.

Single quotes should be used to preserve the literal meanings of all characters, except single quotes. Double quotes preserve the literal meaning of all characters except single quotes, dollar signs, and backslashes. A backslash preserves the literal meaning of the character that follows, except `n`. The newline character (`\n`) has special meaning, and in order to preserve this meaning, it must be enclosed in single quotes, as in `'\n'`. A backslash in double quotes remains literal, unless it precedes the following characters: backlash, single quote, double quotes, dollar sign, and newline. Enclosing characters between `$'` and `'` preserves the literal meaning of all characters, except single quotes and backslashes. `\t` and `\b` also have special meaning and stand for tab and backspace, respectively.

The following session shows a few examples of quoting. `$$` stands for the PID of the process that executes the `echo` command—that is, the current shell process. You can prove it by running the `ps` command and verifying that 61358 is, in fact, the PID of your current Bourne shell process.

```
$ echo '$name * ? \'
$name * ? \
$ echo "$ * \n ?"
$ * \n ?
$ echo "$$ * \n ?"
61358 * \n ?
$ echo "\$ \' \" \\ \\n"
$ \' " \ \ n
$ echo $'a d ? * " $'
a d ? * " $
$ echo $'a \ ? * " $'
Syntax error: Bad escape sequence
$ echo $'a ' ? * " $'
>
```

The following session shows how shell variables can be created and read, and how their values may be changed.

```
$ name=John
$ echo $name
John
$ name=John Doe
Doe: not found
$ name=John date
Sun Aug 3 09:44:35 PKT 2014
$ echo $name
$ name="John Doe"
$ echo $name
John Doe
$ name=John*
$ echo $name
John.Bates.letter John.Johnsen.memo John.email
$ echo "$name"
John*
$ echo "The name $name sounds familiar!"
The name John* sounds familiar!
$ echo \$name
$name
```

```
$ echo '$name'
$name
$
```

If the right-hand side of an assignment statement is not enclosed in quotes and includes spaces, the shell assumes that the second and remaining words in the right-hand side form a command, and tries to execute them. The shell displays an error message if the assumed command does not correspond to a valid command, as shown in the statements `name=John Doe` and `name=John date` in the previous session. Also, after the `name=John*` statement has been executed and `$name` is not quoted in the `echo` command, the shell lists the file names in your present working directory that match `John*`, with `*` considered as the shell metacharacter. In this case, the variable `$name` must be enclosed in quotes to refer to `John*`, as in `echo "$name"`. However, if your current directory does not contain any file that starts with the string `John`, the `echo John*` command displays `John*`.

A command consisting of `$variable` only results in the value of `variable` executed as a shell command. If the value of `variable` comprises a valid command, the expected results are produced. If `variable` does not contain a valid command, the shell, as expected, displays an appropriate error message. The following session makes this point with some examples. The variable used in the session is `command`.

```
$ command=pwd
$ $command
/home/sarwar/unixb3e/examples
$ command=hello
$ $command
hello: not found
$
```

12.3.2 Command Substitution

Command substitution allows you to replace a command with its output. The following is a brief description of command substitution. Note the use of back quotes (also known as *grave accents*) in the first form of the syntax. In our example shell sessions and scripts throughout the book, we will use the first syntax.

SYNTAX

`` `command` ``

Or

`$(command)`

Purpose: Execute command and substitute **`` `command` ``** or **`$(command)`**, if you use this syntax) with the output of the command

The following session illustrates this concept with a few examples. In the first assignment statement, the variable command is assigned a value pwd. In the second and third assignment statements, the output of the pwd command is assigned to the command variable.

```
$ command=pwd
$ echo "The value of command is: $command."
The value of command is: pwd.
$ command=`pwd`
$ echo "The value of command is: $command."
The value of command is: /home/sarwar/unix3e.
$ command=$(pwd)
$ echo "The value of command is: $command."
The value of command is: /home/sarwar/unix3e.
$
```

Command substitution can be specified in any command. For example, in the following session, the output of the date command is substituted for `date` before the echo command is executed.

```
$ echo "The date and time are `date`."
The date and time are Thu Jul 24 23:59:11 PKT 2014.
$
```

We demonstrate the real-world use of command substitution in various ways throughout this chapter and Chapter 13.

The following in-chapter exercises are designed to reinforce the creation and use of shell variables and the concept of command substitution.

EXERCISE 12.3

Assign your full name to a shell variable called myname and echo its value. How did you accomplish the task? Show your work.

EXERCISE 12.4

Assign the output of the echo "Hello, world!" command to the myname variable and then display the value of myname. List the commands that you executed to complete your work.

12.3.3 Exporting Environment

When a variable is created in a shell, subsequent shells do not have automatic access to it. The export command passes the *value* of a variable to subsequent shells. Thus, when a shell script is called and executed in another shell script, it does not get automatic access to the variables defined in the original (i.e., caller) script unless they are explicitly made available to it. The `export` command can be used to pass the value of one or more shell variables to any subsequent script. All read/write shell environment variables are available to every command, script, and subshell, so they are exported at the time they are initialized. The following is a brief description of the `export` command.

SYNTAX

`export [name-list]`

Purpose: Export the names and copies of the current values in **name-list** to every command executed from this point on

The following session presents a simple use of the `export` command. The `name` variable is initialized to **John Doe** and is exported to subsequent commands executed under the current shell and any subshells that run under the current shell.

```
$ name="John Doe"
$ export name
$
```

We now illustrate the concept of exporting shell variables via some simple shell scripts. Consider the following session. Note that the one-line shell script in the **display_name** file displays a null string even though we initialized the `name` variable to `John Doe` just before executing this script. The reason is that the `name` variable is not exported before running the script, and the `name` variable used in the script is local to the script. Because this local variable `name` is uninitialized, the `echo` command displays the null string—the default value of every uninitialized variable.

```
$ cat display_name
echo $name
exit 0
$ name="John Doe"
$ ./display_name
$
```

You can use the `exit` command to transfer control to the calling process—the current shell process in the preceding session. The only argument of the `exit` command is an optional integer number, which is returned to the calling process as the exit status of the terminating process. All UNIX commands return an exit status of zero upon *success* (i.e., if they successfully perform their tasks and terminate normally) and nonzero upon *failure*. The return status value of a command is stored in the read-only environment variable `$?`, which can be checked and/or displayed by the calling process. In shell scripts, the status of a command is commonly checked and subsequent action taken. We show the use of `$?` in some shell scripts later in the chapter. When the `exit` command is executed without an argument, the UNIX kernel sets the return status value for the script.

In the following session, the `name` variable is exported after it is initialized, thus making it available to the **display_name** script. Because the `name` variable is exported before the **display_name** script is executed, the script displays `John Doe` and not a null string as was the result of its execution in the previous session. The session also shows that the return status of the **display_name** script is 0. Note that you can combine the initialization of a variable and exporting it in one command as in `export name="John Doe"`.

```
$ name="John Doe"
$ ./export name
$ ./display_name
John Doe
$ echo $?
0
$
```

We now show that a copy of an exported variable's value is passed to any subsequent command. In other words, a command has access to the value of the exported variable only; it cannot assign a new value to the variable. Consider the script in the **export_demo** file.

```
$ cat export_demo
#!/bin/sh
name="John Doe"
export name
display_change_name
display_name
exit 0
$ cat display_change_name
#!/bin/sh
echo $name
name="Plain Jane"
echo $name
exit 0
$ ./export_demo
John Doe
Plain Jane
John Doe
$
```

When the **export_demo** script is invoked, the `name` variable is set to `John Doe` and exported so that it becomes part of the environment of all the commands that execute under **export_demo**. The first `echo` command in the **display_change_name** script displays `John Doe` as the value of the exported variable (nonlocal) `name`. It then initializes a local variable, name, to Plain Jane. The second echo command therefore echoes the current value of the local variable name and displays Plain Jane. When the **display_change_name** script has finished its execution, the **display_name** script executes and displays the value of the exported (nonlocal) name, thus displaying John Doe.

12.3.4 Resetting Variables

A variable retains its value as long as the script in which it is initialized is running. You can reset the value of a variable to null (the default initial value of all variables) by either explicitly initializing it to null or by using the unset command. The following is a brief description of this command.

SYNTAX

```
unset [name-list]
```

Purpose: Reset or remove the variable or function corresponding to the names in **name-list**, where **name-list** is a list of names separated by spaces

We discuss functions in the Bourne shell in Chapter 13, so we limit the discussion of the unset command here to variables only. The following session shows a simple use of the command. The variables name and place are set to John and Corvallis, respectively, and the echo command displays the values of these variables. The unset command resets name to null. Thus, the echo "$name" command displays a null string (a blank line).

```
$ name=John place=Corvallis
$ echo "$name $place"
John Corvallis
$ unset name
$ echo "$name"
$ echo "$place"
Corvallis
$
```

The following command removes the variables name and place.

```
$ unset name place
$
```

Another way to reset a variable is to assign it explicitly a null value by assigning it no value and simply hitting <Enter> after the = sign, as in

```
$ country=
$ echo "$country"
$
```

12.3.5 Creating Read-Only Defined Variables

When programming, you sometimes need to use constants. You can use literal constants, but using symbolic constants is good programming practice, primarily because it makes your code more readable. Another reason for using symbolic names is that a constant used at various places in code might need to be changed. With a symbolic constant, the change is made at one place only, but a literal constant must be changed every place it was used. A symbolic constant can be created in the Bourne shell by initializing a variable with the desired value and making it read only by using the `readonly` command. This command is rarely used in shell scripts, but we discuss it briefly for the sake of complete coverage of shell variables. The following is a brief description of the command.

SYNTAX

`readonly [name-list]`

Purpose: Prevent assignment of new values to the variables in **name-list**

In the following session, the `name` and `place` variables are made read only after initializing them with `John` and `Ames`, respectively. Once they have become read only, assignment to either variable fails.

```
$ name=John
$ place=Ames
$ readonly name place
$ echo "$name $place"
John Ames
$ name=Art
name: is read only
$ place="Ann Arbor"
place: is read only
$
```

When the `readonly` command is executed without arguments, it displays all read-only variables, as in the following session:

```
$ readonly
name
place
$
```

On some UNIX systems, the output of the `readonly` command shows the list of read-only variables along with the values of these variables.

12.3.6 Reading from Standard Input

So far, we have shown how you can assign values to shell variables statically at the command line level or by using the assignment statement in your programs. If you want to write an interactive shell script that prompts the user for keyboard input, you need to use the `read` command to store the user input in a shell variable. This command allows you to read one line of standard input. The following is a brief description of the command.

SYNTAX

`read variable-list`

Purpose: Read one line from standard input and assign words in the line to variables in **variable-list**

A line is read in the form of words separated by white spaces (`<Space>` or `<Tab>` characters, depending on the value of the shell environment variable `IFS`). The words are assigned to the variables in the order of their occurrence, from left to right. If the number of words in the line is greater than the number of variables in **variable-list**, the last variable is assigned the extra words. If the number of words in a line is less than the number of variables, the remaining variables are reset to null.

We illustrate the semantics of the `read` command by way of the following script in the **read_demo** file.

```
$ cat read_demo
#!/bin/sh
echo -n "Enter input: "
read line
echo "You entered: $line"
echo -n "Enter another line: "
read word1 word2 word3
echo "The first word is: $word1"
echo "The second word is: $word2"
echo "The rest of the line is: $word3"
exit 0
$
```

We now show how the input that you enter from the keyboard is read by the `read` command in that script. In the following run, you enter the same input: `UNIX rules the network computing world!`. The first `read` command takes the whole input and puts it in the shell variable `line` without the newline character. In the second `read` command, the first word of your input is assigned to the variable `word1`, the second word is assigned to the variable `word2`, and the rest of the line (without the newline character) is assigned to the variable `word3`. The outputs of the `echo` commands for displaying the values of these variables confirm this point.

```
$ ./read_demo
Enter input: UNIX rules the network computing world!
You entered: UNIX rules the network computing world!
Enter another line: UNIX rules the network computing world!
The first word is: UNIX The second word is: rules
The rest of the line is: the network computing world!
$
```

The -n option used in the two echo commands is used to force the cursor to stay at the same line after the echo command has displayed the quoted text. If you do not use this character, the cursor moves to the next line, which is what you like to see happen while displaying information and error messages. However, when you prompt the user for keyboard input, you should keep the cursor in front of the prompt for a user-friendlier interface.

On BSD and System V compatible UNIX systems, including PC-PSD, several special characters can be used in the Bourne shell echo -e command as control characters. These characters start with a backslash (\). Thus, for example, the echo -n "Enter input: " command in the **read_demo** script may be replaced with echo -e "Enter input: \c". Some of the control characters, along with their meanings, are listed in Table 12.3. Note that on some UNIX systems, you may have to use double backslash (\\) instead of single backslash for the special characters to work. For example, use \\c instead of \c to keep the cursor on the same line.

In the following in-chapter exercises, you will use the read and export commands to practice reading from **stdin** in shell scripts and exporting variables to child processes.

EXERCISE 12.5

Give commands for reading a value into the myname variable from the keyboard and exporting it so that commands executed in any child shell have access to the variable.

TABLE 12.3 Special Characters for the echo -e Command

Character	Meaning
\b	Backspace
\c	Prints line without moving cursor to next line
\f	Form feed
\n	Newline (move cursor to next line)
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\\	Backslash (escape special meaning of backslash)
\0N	Character whose ASCII number is octal N (it is 0 and not o)

Note: You might have to use \\ instead of \ on some UNIX systems.

EXERCISE 12.6

Copy the value myname variable to another variable, anyname. Make the anyname variable read only and unset both the myname and anyname variables. What happened? Show all your work.

12.4 PASSING ARGUMENTS TO SHELL SCRIPTS

In this section, we describe how command line arguments can be passed to shell scripts and manipulated by them. As we discussed in Section 12.3, you can pass command line arguments, or *positional parameters*, to a shell script. The values of these arguments, starting with the first argument, are stored in variables $1, $2, $3, $4, and so on, respectively. The variable name $0 contains the name of the script file (i.e., the command name). On some UNIX systems, up to only the first nine arguments are stored in these variables. You can use the names of these variables to refer to the values of these arguments. If the positional argument that you refer to is not passed an argument, it has a value of null. The environment variable $# contains the total number of arguments passed in an execution of a script. The variables $* and $@ both contain the values of all of the arguments, but $@ has each individual argument in quotes if it is used as "$@". The shell script in the **cmdargs_demo** file shows how you can use these variables.

```
$ cat cmdargs_demo
#!/bin/sh
echo "The command name is: $0."
echo "The number of command line arguments passed as parameters is
$#."
echo "The values of the command line arguments are: $1 $2 $3 $4 $5
$6 $7 $8 $9 $10 $11"
echo "Another way to display the values of all of the arguments
is: $@."
echo "Yet another way is: $*."
exit 0
$ ./cmdargs_demo a b c d e f g h i j k l m n
The command name is: cmdargs_demo.
The number of command line arguments passed as parameters is 9.
The values of the command line arguments are: a b c d e f g h i j k.
Another way to display the values of all of the arguments: a b c d
e f g h i j k l m n.
Yet another way is: a b c d e f g h i j k l m n.
$ ./cmdargs_demo One Two 3 Four 5 6
The command name is: cmdargs_demo.
The number of command line arguments passed as parameters is 6.
The values of the command line arguments are: One Two 3 Four 5 6.
Another way to display the values of all of the arguments: One Two
3 Four 5 6.
Yet another way is: One Two 3 Four 5 6.
$
```

On systems where the shell maintains up to nine positional arguments at a time in the shell variables `$1– $9`, you can write scripts that can accept and process more than nine arguments. To do so, use the `shift` command. By default, the command shifts the command line arguments to the left by one position, making `$2` become `$1`, `$3` become `$2`, and so on. The first argument, `$1`, is shifted out. Once shifted, the arguments cannot be restored to their original values. The number of positions to be shifted can be more than one and specified as an argument to the command. The following is a brief description of the command.

SYNTAX

`shift [N]`

Purpose: Shift the command line arguments N positions to the left

The script in the **shift_demo** file shows the semantics of the `shift` command. The first `shift` command shifts the first argument out and the remaining arguments to the left by one position. The second `shift` command shifts the current arguments to the left by three positions. The three `echo` commands are used to display the current value of the program name (`$0`), the values of all positional parameters (`$@`), and the values of the first three positional parameters, respectively. The results of execution of the script are obvious.

```
$ cat shift_demo
#!/bin/sh
echo "The program name is $0."
echo "The arguments are: $@"
echo "The first three arguments are: $1 $2 $3"
shift
echo "The program name is $0."
echo "The arguments are: $@"
echo "The first three arguments are: $1 $2 $3"
shift 3
echo "The program name is $0."
echo "The arguments are: $@"
echo "The first three arguments are: $1 $2 $3"
exit 0
$ ./shift_demo 1 2 3 4 5 6 7 8 9 10 11 12
The program name is shift_demo.
The arguments are: 1 2 3 4 5 6 7 8 9 10 11 12
The first three arguments are: 1 2 3
The program name is shift_demo.
The arguments are: 2 3 4 5 6 7 8 9 10 11 12
The first three arguments are: 2 3 4
The program name is shift_demo.
The arguments are: 5 6 7 8 9 10 11 12
The first three arguments are: 5 6 7
$
```

The `set` command can be used to alter the values of positional arguments. The most effective use of this command is in conjunction with command substitution. The following is a brief description of the command.

SYNTAX

`set [options] [argument-list]`

Purpose: Set values of the positional arguments to the arguments in **argument-list**; when executed without an argument, the `set` command displays names of all shell variables and their current values (as shown in Section 12.3)

The following session involves a simple interactive use of the `set` command. The `date` command is executed to show that the output has six fields. The `set 'date'` command sets the positional parameters to the output of the `date` command. In particular, `$1` is set to `Sat`, `$2` to `Jul`, `$3` to `26`, `$4` to `16:29:35`, `$5` to `PKT`, and `$6` to `2014`. The `echo "$@"` command displays the values of all positional arguments. The third `echo` command displays the date in a commonly used form.

```
$ date
Sat Jul 26 16:29:35 PKT 2014
$ set 'date'
$ echo "$@"
Sat Jul 26 16:29:55 PKT 2014
$ echo "$2 $3, $6"
Jul 26, 2014
$
```

An option commonly used with the `set` command is `--`. It is used to inform the `set` command that, if the first argument starts with a `-`, it should not be considered an option for the `set` command. The script in **set_demo** shows another use of the command. When the script is run with a file argument, it generates a line that contains the file name, the file's inode number, and the file size (in bytes). Note that on some systems, the size variable needs to be set to `$5`. The `set` command is used to assign the output of the `ls -l` command as the new values of the positional arguments `$1`–`$9`. If you do not remember the format of the output of the `ls -l` command, we suggest you run this command on a file before studying the code. The first string in the output starts with a – for an ordinary file. Thus, with the use of the – option the `set` command does not regard – (the first character in the permissions string) as the start of an option for the command and deals with it literally.

```
$ cat set_demo
#!/bin/sh
filename="$1"
set -- `ls -l $filename`
perms="$1"
size="$5"
set `ls -i $filename`
inode="$1"
echo "File Name:        $filename"
echo "Inode Number:     $inode"
echo "Permissions:      $perms"
echo "Size (bytes):     $size"
exit 0
$ ./set_demo lab1
File Name:  lab1
Inode Number:     679
Permissions:      -rwxr--r--
Size (bytes):     243
$
```

In the following in-chapter exercises, you will use the `set` and `shift` commands to reinforce the use of command line arguments and their processing.

EXERCISE 12.7

Write a shell script that displays all command line arguments, shifts them to the left by two positions, and redisplays them. Show the script along with a few sample runs.

EXERCISE 12.8

Update the shell script in Exercise 12.7 so that, after accomplishing the original task, it sets the positional arguments to the output of the `who | head -1` command and displays the positional arguments again.

12.5 COMMENTS AND PROGRAM HEADERS

You should develop the habit of putting comments in your programs to describe the purpose of a particular series of commands. At times, you should even briefly describe the purpose of a variable or assignment statement. Also, you should use a program header for every shell script that you write. These are simply good software engineering practices. A *program header* is a set of introductory comments used to explain the script. Program header and in-code comments help a programmer who has been assigned the task of maintaining (i.e., modifying or enhancing) your code to understand it quickly. They also help you understand your own code, in particular, if you reread it after some period of time. Long ago, putting comments in the program code or creating separate documentation for programs was not a common practice. Such programs, when inherited by a programmer or a team, are very difficult to understand and maintain, and are commonly known as *legacy code*. You may find different definitions for legacy code in the literature.

A good program header must contain at least the following items. In addition, you can insert any other items that you believe to be important or are commonly used in your organization or group as part of its coding rules.

1. Name of the file containing the script
2. Name of the author
3. Date written
4. Date last modified
5. Purpose of the script (in one or two lines)
6. A brief description of the algorithm used to implement the solution to the problem at hand

A comment line, including every line in the program header, must start with the number sign (#), as in

```
# This is a comment line.
```

However, a comment does not have to start at a new line; it can follow a command, as in:

```
set -- 'ls -l lab1'   # Assign new values to positional parameters
and
                      # if the first argument starts with a -, do
not
                      # consider it an option for the set command.
                      # This is to handle the output of the ls -l
                      # command if lab1 is an ordinary file.
```

The following is a sample header for the **set_demo** script.

```
# File Name:            ~/Bourne/examples/set_demo
# Author:               Syed Mansoor Sarwar
# Date Written:         August 10, 1999
# Modified:             May 21, 2004 (by the original author)
# Date Last Modified:   July 27, 2014 (by the original author)
# Purpose:              To illustrate how the set command works
# Brief Description:    The script runs with a filename as the
                        only command
#                       line argument, saves the filename, runs
                        the set command
```

```
#                       to assign output of ls -il command to
                        positional
#                       arguments ($1–$9), and displays file
                        name,its
#                       inode number, file permissions, and its
                        size in bytes.
```

We do not show the program headers for all the sample scripts in this textbook for the sake of brevity.

12.6 PROGRAM CONTROL FLOW COMMANDS

The program control flow commands/statements are used to determine the sequence in which statements in a shell script execute. There are three basic types of statements for controlling the flow of a script: two-way branching, multiway branching, and repetitive execution of one or more commands. The Bourne shell statement for two-way branching is `if`, the statements for multiway branching are `if` and `case`, and the statements for repetitive execution of some code are `for`, `while`, and `until`.

12.6.1 The `if-then-elif-else-fi` Statement

The most basic form of the `if` statement is used for one-way branching, but the statement can also be used for two-way and multiway branching. The following is the syntax and a brief description of the multiway `if` statement. The words in monospace type are *keywords* and must be used as shown in the syntax. Everything in brackets is optional. All the command lists are designed to enable you to accomplish the task at hand.

SYNTAX

```
if expression
    then
        [elif expression
        then
            then-command-list]
        ...
        [else
            else-command-list]
fi
```

Purpose: To implement two-way or multiway branching

Here, **expression** is a list of commands. The execution of commands in **expression** returns a status of true (success) or false (failure). We discuss three versions of the **`if`** statement that together comprise the statement's complete syntax and semantics. The first version of the **`if`** statement is without any optional features, which results in the syntax for the statement that is commonly used for one-way branching.

SYNTAX

```
if expression
    then
        then-commands
fi
```

Purpose: To implement two-way branching

If **expression** is true, the **then-commands** are executed; otherwise, the command after **`fi`** is executed. The semantics of the statement are illustrated in Figure 12.1.

The expression can be evaluated with the `test` command. It evaluates an expression and returns true or false. The command has two syntaxes: One uses the keyword `test` and the other uses brackets. The following is a brief description of the command.

SYNTAX

`test` **[expression]**

Or

[[expression]]

Purpose: To evaluate **expression** and return true or false status

An important point about this second syntax is that the normal (inside) brackets indicate an optional expression and that the monospace (outside) brackets are required because

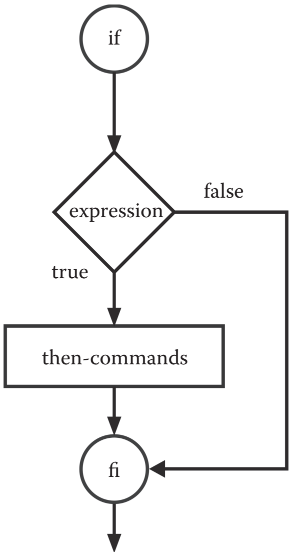


FIGURE 12.1 Semantics of the `if-then-fi` statement.

they comprise the test statement. Also, at least one space is required before and after an operator, a parenthesis, a bracket, or an operand. If you need to continue a test expression to the next line, you must use a backslash (\) before hitting <Enter> so that the shell does not treat the next line as a separate command. Recall that \ is a shell metacharacter. We demonstrate use of the test command in the first session but then use the simpler syntax of [].

The test command supports many operators for testing files and integers, testing and comparing strings, and logically connecting two or more expressions to form complex expressions. Table 12.4 describes the meanings of the operators supported by the test command on most UNIX systems.

We use the if statement to modify the script in the **set_demo** file so that it takes one command line argument only and checks on whether the argument is a file or a directory. The script returns an error message if the script is run with no or more than one command line argument, or if the command line argument is not an ordinary file. The name of the script file is **if_demo1**.

TABLE 12.4 Operators for the test Command

File Testing		Integer Testing		String Testing	
Expression	**Return Value**	**Expression**	**Return Value**	**Expression**	**Return Value**
`-d file`	True if **file** is a directory	`int1 -eq int2`	True if **int1**and **int2** are equal	`str`	True if **str** is not an empty string
`-f file`	True if **file** is an ordinary file	`int1 -ge int2`	True if **int1** is greater than or equal to **int2**	`str1 = str2`	True if **str1** and **str2** are the same
`-r file`	True if **file** is readable	`int1 -gt int2`	True if **int1** is greater than **int2**	`str1 != str2`	True if **str1** and **str2** are not the same
`-s file`	True if length of **file** is nonzero	`int1 -le int2`	True if **int1** is less than or equal to **int2**	`-n str`	True if the length of **str** is greater than zero
`-t [filedes]`	True if file descriptor **filedes** is associated with the terminal	`int1 -lt int2`	True if **int1** is less than **int2**	`-z str`	True if the length of **str** is zero
`-w file`	True if **file** is writable	`int1 -ne int2`	True if **int1** is not equal to **int2**		
`-x file`	True if **file** is executable				

Operators for Forming Complex Expressions

`!`	Logical NOT operator: true if the following expression is false	**(expression)**	Parentheses for grouping expressions; at least one space before and one after each parenthesis
`-a`	Logical AND operator: true if the previous (left) and next (right) expressions are true	`-o`	Logical OR operator: true if the previous (left) or next (right) expression is true

```
$ cat if_demo1
#!/bin/sh
if test $# -eq 0
    then
        echo "Usage: $0 ordinary_file"
        exit 1
fi
if test $# -gt 1
    then
        echo "Usage: $0 ordinary_file"
        exit 1
fi
if test -f "$1"
    then
        filename=$1
        set -- `ls -l $filename`
        perms="$1"
        size="$5"
        set `ls -i $filename`
        inode="$1"
        echo "File Name:        $filename"
        echo "Inode Number:     $inode"
        echo "Permissions:      $perms"
        echo "Size (bytes):     $size"
        exit 0
fi
echo "$0: argument must be an ordinary file"
exit 1
$ ./if_demo1
Usage: if_demo1 ordinary_file
$ ./if_demo1 lab1 lab4
Usage: if_demo1 ordinary_file
$ ./if_demo1 dir1
if_demo1: argument must be an ordinary file
$ ./if_demo1 lab1
File Name:        lab1
Inode Number:     679
Permissions:      -rwxr--r--
Size (bytes):     243
$
```

In the preceding script, the first `if` statement displays an error message and exits the program if you run the script without any command line argument. The second `if` statement displays an error message and exits the program if you run the script with more than one argument. The third `if` statement is executed if conditions for the first two are false—that is, if you run the script with one argument only. This `if` statement produces the desired results if the command line argument is an ordinary file. If the passed argument is not an ordinary file, the condition for the third `if` statement is false and the error message `if _ demo1: argument must be an ordinary file` is displayed. The `exit` command is used to take control out of the script. We normally use `1` as the argument (i.e., exit status) of `exit` when a script is to be terminated because of an erroneous condition. The exit status `0` is used for normal termination of the script.

An important practice in script writing is to correctly indent the commands/statements in it. Proper indentation of programs enhances their readability and makes them easier to understand and maintain. However, the white spacing neither impacts the syntactic correctness of the program nor its efficiency when executed. Note the indentation style used in our sample scripts and follow it when you write your own scripts.

We now discuss the second version of the `if` statement, which also allows two-way branching. The following is a brief description of the statement.

SYNTAX

```
if expression
    then
        then-commands
    else
        else-commands
fi
```

Purpose: To implement two-way branching

If **expression** is true, the commands in **then-commands** are executed; otherwise, the commands in **else-commands** are executed, followed by the execution of the command after **fi**. The semantics of the statement are depicted in Figure 12.2.

Next, we rewrite the **if_demo1** program, using the `if-then-else-fi` statement, and use the alternative syntax for the `test` command. The resulting script is in the **if_demo2** file, as shown in the following session. Note that the program looks cleaner and more readable.

```
$ cat if_demo2
#!/bin/sh
if [ $# -eq 0 ]
   then
      echo "Usage: $0 ordinary_file"
      exit 1
fi
if [ $# -gt 1 ]
   then
      echo "Usage: $0 ordinary_file"
      exit 1
fi
if [ -f "$1" ]
```

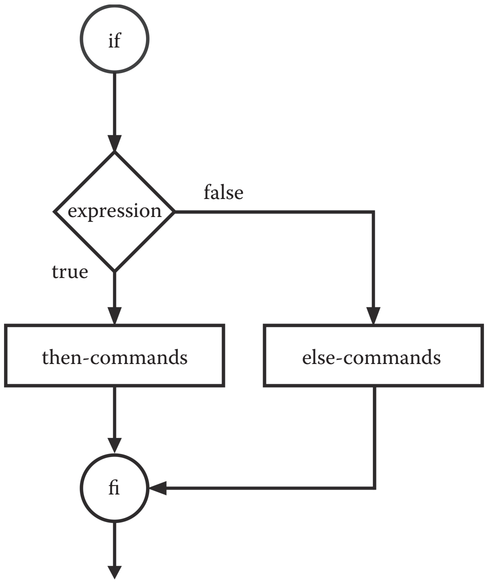


FIGURE 12.2 Semantics of the `if-then-else-fi` statement.

```
   then
      filename=$1
      set -- `ls -l $filename`
      perms="$1"
      size="$5"
      set `ls -i $filename`
      inode="$1"
      echo "File Name:        $filename"
      echo "Inode Number:     $inode"
      echo "Permissions:      $perms"
      echo "Size (bytes):     $size"
      exit 0
   else
      echo "$0: argument must be an ordinary file"
      exit 1
fi
exit 0
$
```

Finally, we discuss the third version of the `if` statement, which is used to implement multiway branching. The following is a brief description of the statement.

SYNTAX

```
if expression1
    then
        then-commands
    elif expression2
            elif1-commands
     elif  expression3
            elif2-commands
     ...
     else
            else-commands
fi
```

Purpose: To implement multiway branching

If **expression1** is true, the commands in **then-commands** are executed. If **expression1** is false, **expression2** is evaluated, and if it is true, the commands in **elif1-commands** are executed. If **expression2** is also false, **expression3** is evaluated. If **expression3** is true, the commands in **elif2-commands** are executed. If **expression3** is also false, the commands in **else-commands** are executed. The execution of any command list is followed by the execution of the command after `fi`. You can use any number of `elif`s in an `if` statement to implement multiway branching. The semantics of the statement are depicted in Figure 12.3.

We modify the script in the **if_demo** file so that, if the command line argument is a directory, the program displays the number of files and subdirectories in the directory, excluding the hidden files. In addition, the program ensures that the command line argument is an existing file or directory in the current directory before processing it. In addition to using two `if` statement, we also use the `if-then-elif-else-fi` statement in

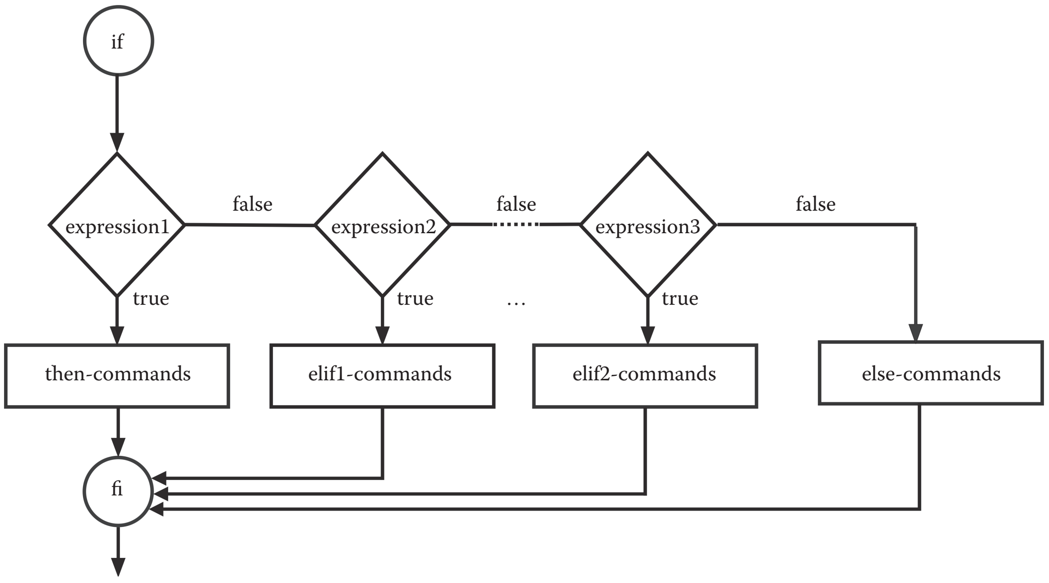


FIGURE 12.3 Semantics of the `if-then-elif-else-fi` statement.

the implementation. The resulting script is in the **if_demo3** file, as shown in the following session.

```
$ cat if_demo3
#!/bin/sh
if [ $# -eq 0 ]
   then
        echo "Usage: $0 file"
        exit 1
   elif [ $# -gt 1 ]
   then
        echo "Usage: $0 file"
        exit 1
   elif [ -d "$1" ]
        then
           nfiles='ls "$1" | wc -w'
           echo "The number of files in the $1 directory is
$nfiles"
           exit 0
   else
        ls "$1" 2> /dev/null | grep "$1" 2> /dev/null 1>&2
        if [ $? -ne 0 ]
           then
                echo "$1: not found"
                exit 1
        fi
        if [ -f "$1" ]
           then
                filename=$1
                set -- 'ls -l $filename'
                perms="$1"
                size="$5"
                set 'ls -i $filename'
                inode="$1"
                echo "File Name:        $filename"
                echo "Inode Number:     $inode"
                echo "Permissions:      $perms"
                echo "Size (bytes):     $size"
                exit 0
           else
                echo "$0: argument must be an ordinary file or
directory"
                exit 1
        fi
fi
$ ./if_demo3 /bin/ls
File Name:        /bin/ls
```

```
Inode Number:     12035
Permissions:      -r-xr-xr-x
Size (bytes):     29512
$ ./if_demo3 lab2
lab2: not found
$ ./if_demo3 lab1 lab2
Usage: if_demo3 file
$ ./if_demo3 ~
The number of files in the /home/sarwar directory is 14
$ ./if_demo3 lab1
File Name:        lab1
Inode Number:     679
Permissions:      -rwxr--r--
Size (bytes):     243
$
```

If the argument is a directory, the number of files in it, excluding directories and hidden files, is saved in the `nfiles` variable. The command `ls "$1" 2> /dev/null | grep "$1" 2>/dev/null 1>&2` is executed to check whether the file passed as the command line argument exists. The standard error is redirected to **/dev/null** (the UNIX black hole), and standard output is redirected to standard error by using `1>&2`. Thus, the command does not produce any output or error messages; its only purpose is to set the command's return status value in `$?`. If the command line argument exists, the `ls` command is successful and `$?` contains `0`; otherwise, it contains a nonzero value. If the command line argument is a file, the required file-related data is displayed. Note the use of command substitution and pipe for setting the value of the `nfiles` variable.

In the following in-chapter exercises, you will practice the use of the `if` statement, command substitution, and manipulation of positional parameters.

EXERCISE 12.9

Create the **if_demo2** script file and run it with no argument, more than one argument, and one argument only. While running the script with one argument, use a directory as the argument. What happens? Does the output make sense?

EXERCISE 12.10

Write a shell script whose single command line argument is a file. If you run the program with an ordinary file, the program displays the owner's name and last update time for the file. If the program is run with more than one argument, it generates meaningful error messages.

12.6.2 The `for` Statement

The `for` statement is the first of three statements that are available in the Bourne shell for repetitive execution of a block of commands in a shell script. These statements are commonly known as *loops*. The following is a brief description of the statement.

SYNTAX

```
for variable [in argument-list]
do
    command-list
done
```

Purpose: To execute commands in **command-list** as many times as the number of items in the **argument-list**; without the optional part in **argument-list** the arguments are supplied at the command line

The items in **argument-list** are assigned to **variable** one by one, and the commands in **command-list**, also known as the body of the loop, are executed for every assignment. This process allows the execution of commands in **command-list** as many times as the number of items in **argument-list**. Figure 12.4 illustrates the semantics of the `for` command. Items may be numbers of words.

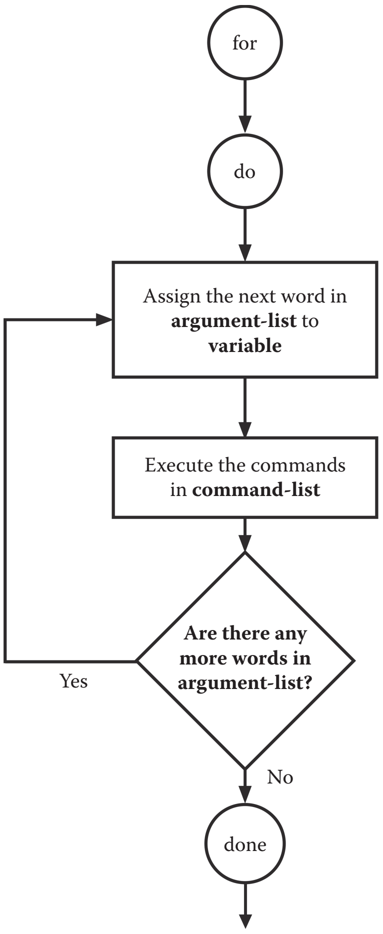

for
do
Assign the next word in argument-list to variable
Execute the commands in command-list
Are there any more words in argument-list?
Yes
No
done

FIGURE 12.4 Semantics of the `for` statement.

The following script in the **for_demo1** file shows the use of the for command with optional arguments. The variable people is assigned the words in **argument-list** one by one, and each time the value of the variable is echoed until no word remains in the list. At that time, control comes out of the for statement, and the command following done is executed. Then the code following the for statement, the exit 0 statement only in this case, is executed.

```
$ cat for_demo1
#!/bin/sh
for people in Debbie Jamie John Kitty Kuhn Shah
do
       echo "$people"
done
exit 0
$ ./for_demo1
Debbie
Jamie
John
Kitty
Kuhn
Shah
$
```

The following script in the **user_info** file takes a list of existing (i.e., valid) login names as command line arguments and displays each login name and the full name of the user who owns the login name, one per login. In the sample run, the first value of the user variable is **dheckman**. The echo command displays dheckman: followed by a <Tab>, and the cursor stays at the current line. The first grep command is used to check if **dheckman** has an entry in the **/etc/passwd** file. If the answer is no, the process is repeated for the second command line argument. If **dheckman** is found, the second grep command searches the **/etc/passwd** file for the login name **dheckman** and pipes it to the cut command, which displays the fifth field in the **/etc/passwd** line for **dheckman** (his full name). The process is repeated for the remaining two login names (**ghacker** and **msarwar**). No user is left in the list passed at the command line, so control comes out of the for statement and the exit 0 command is executed to transfer control back to shell. The command substitution "^"`echo $user":"` in the grep command can be replaced by "^"$user":".

```
$ cat user_info
#!/bin/sh
for user
do
# Don't display anything if a login name is not found in /etc/
passwd
   grep "^"`echo $user":"` /etc/passwd 1> /dev/null 2>&1
   if [ $? -eq 0 ]
```

```
        then
           echo -n "$user:      "
           grep "^"`echo $user":"` /etc/passwd | cut -f5 -d':'
   fi
done
exit 0
$ ./user_info dheckman ghacker sarwar
dheckman: Dennis R. Heckman
ghacker:    George Hacker
sarwar:     Syed Mansoor Sarwar
$
```

12.6.3 The `while` Statement

The `while` statement, also known as the `while` loop, allows repeated execution of a block of code based on the condition of an expression. The following is a brief description of the statement.

SYNTAX

```
while expression
do
    command-list
done
```

Purpose: To execute commands in **command-list** as long as expression evaluates to true

The **expression** is evaluated and, if the result of this evaluation is true, the commands in **command-list** are executed and **expression** is evaluated again. This sequence of expression evaluation and execution of **command-list**, known as one iteration, is repeated until the **expression** evaluates to false. At that time, control comes out of the **`while`** statement and the statement following **`done`** is executed. Figure 12.5 depicts the semantics of the `while` statement.

The variables and/or conditions in the expression that result in a true value must be correctly manipulated in the commands in **command-list** for well-behaved loops—that is, loops that eventually terminate and allow execution of the rest of the code in a script. Loops in which the expression always evaluates to true are known as *nonterminating*, or *infinite*, loops. Infinite loops, usually a result of poor design and/or programming, are undesirable because they continuously use CPU time without accomplishing any useful task. However, some applications do require infinite loops. For example, all the servers for Internet services such as the HTTP service (which allows us to browse webpages on the Internet) are programs that run indefinitely, waiting for client requests. In case of the HTTP service, for example, the client requests come through Web browsers (i.e., HTTP clients) such as Mozilla Firefox. Once a server has received a client request, it processes it, sends a response to the client, and waits for another client request. The only way to terminate a process with an infinite loop is to kill it by using the `kill` command. Or, if the process is running in the foreground, pressing

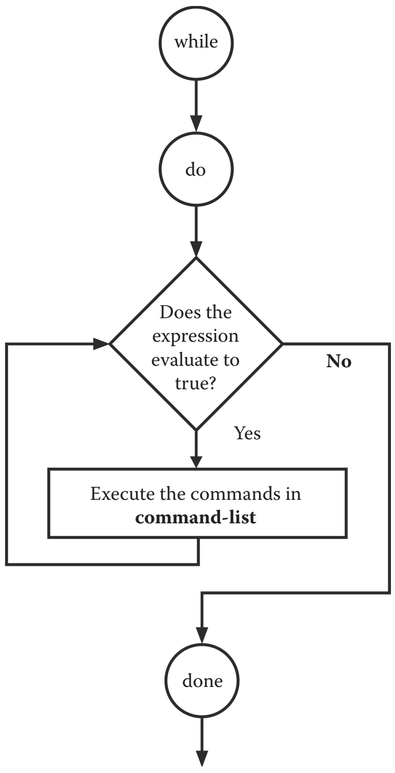

while
do
Does the expression evaluate to true?
No
Yes
Execute the commands in **command-list**
done

FIGURE 12.5 Semantics of the `while` statement.

`<Ctrl+C>` would also do the trick, unless the process is designed to ignore `<Ctrl+C>`. In that case, you need to put the process in the background by pressing `<Ctrl+Z>` and use the `kill -9` command to terminate it. See Chapter 10 for details on processes.

The script in the **while_demo** file shows a simple use of the `while` loop. When you run this script, the `secretcode` variable is initialized to `agent007` and you are prompted to make a guess. Your guess is stored in a local variable `yourguess`. If your guess is not `agent007`, the condition for the `while` loop is true and the commands between `do` and `done` are executed. This program displays a message tactfully informing you of your failure and prompts you for another guess. Your guess is again stored in the `yourguess` variable, and the condition for the loop is tested. This process continues until you enter `agent007` as your guess. At which time, the condition for the loop becomes false and the control comes out of the `while` statement. The `echo` command following `done` executes, congratulating you for being part of a great gene pool!

```
$ cat while_demo
#!/bin/sh
secretcode=agent007
echo "Guess the code!"
echo -e "Enter your guess: \c"
read yourguess
```

```
while [ "$secretcode" != "$yourguess" ]
do
        echo "Good guess but wrong. Try again!"
        echo -e "Enter your guess: \c"
        read yourguess
done
echo "Wow! You are a genius!!"
exit 0
$ ./while_demo
Guess the code!
Enter your guess: star wars
Good guess but wrong. Try again! Enter your guess: columbo
Good guess but wrong. Try again! Enter your guess: agent007
Wow! You are a genius!!
$
```

12.6.4 The `until` Statement

The syntax of the `until` statement is similar to that of the `while` statement, but its semantics are different. Whereas in the `while` statement the loop body executes as long as the expression evaluates to true, in the `until` statement the loop body executes as long as the expression evaluates to false. The following is a brief description of the statement.

SYNTAX

```
until expression
do
    command-list
done
```

Purpose: To execute commands in **command-list** as long as expression evaluates to false

Figure 12.6 illustrates the semantics of the `until` statement. The code in the **until_demo** file performs the same task that the script in the **while_demo** file does (see Section 12.6.3), but it uses the `until` statement instead of the `while` statement. Thus, the code between do and done (the loop body) is executed for as long as your guess is not `agent007`.

```
$ cat until_demo
#!/bin/sh
secretcode=agent007
echo "Guess the code!"
echo -n "Enter your guess: "
read yourguess
until [ "$secretcode" = "$yourguess" ]
do
      echo "Good guess but wrong. Try again!"
      echo -n "Enter your guess: "
```

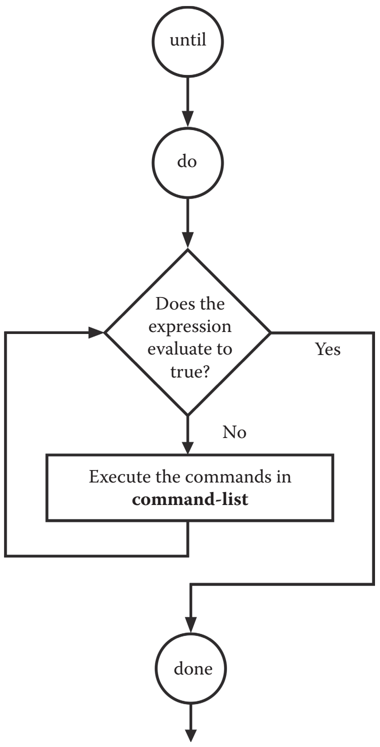

until
do
Does the expression evaluate to true?
Yes
No
Execute the commands in **command-list**
done

FIGURE 12.6 Semantics of the `until` statement.

```
      read yourguess
done
echo "Wow! You are a genius!!" exit 0
$ ./until_demo
Guess the code!
Enter your guess: Inspector Gadget
Good guess but wrong. Try again!
Enter your guess: Peter Sellers
Good guess but wrong. Try again!
Enter your guess: agent007
Wow! You are a genius!!
$
$
```

12.6.5 The `break` and `continue` Commands

The `break` and `continue` commands can be used to interrupt the sequential execution of the loop body. The `break` command transfers control to the command following `done`, thus terminating the loop prematurely. The `continue` command transfers control to `done`, which results in the evaluation of the condition again and hence, continuation of the loop. In both cases, the commands in the loop body following these statements are not executed. Thus, these statements are almost always used in an `if` or `if-else` setting. Figure 12.7 illustrates the semantics of these commands.

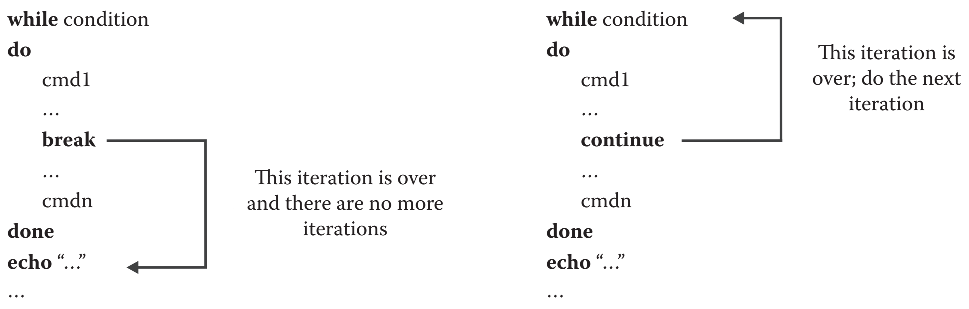

while condition
do
cmd1
…
break
…
cmdn
done
echo "…"
…
This iteration is over and there are no more iterations
while condition
do
cmd1
…
continue
…
cmdn
done
echo "…"
…
This iteration is over; do the next iteration

FIGURE 12.7 Semantics of the `break` and `continue` commands.

In the following in-chapter exercises, you will write shell scripts with loops by using the `for`, `while`, and `until` statements.

EXERCISE 12.11

Write a shell script that takes a list of host names on your network as command line arguments and displays whether the hosts are up or down. Use the `ping` command to display the status of a host and the `for` statement to process all host names.

EXERCISE 12.12

Rewrite the script in Exercise 12.11, using the `while` statement. Rewrite it again, using the `until` statement.

12.6.6 The `case` Statement

The `case` statement provides a mechanism for multiway branching similar to a nested `if` statement. However, the structure provided by the `case` statement is more readable. You would use the `case` statement when you can—that is, when you are testing a single variable to several distinct patterns. You would not use it when you want to test more than one variable. The following is a brief description of the statement.

SYNTAX

```
case test-string in
pattern1)       command-list1
                ;;
pattern2)       command-list2
                ;;
...
patternN)       command-listN
                ;;
esac
```

Purpose: To implement multiway branching like a nested `if`

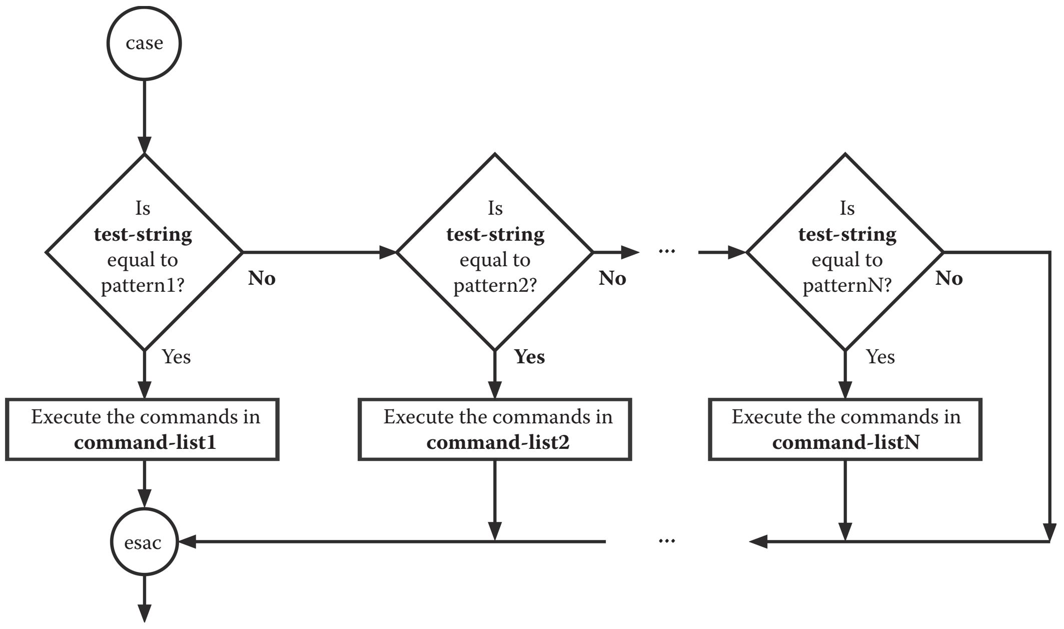

case
Is test-string equal to pattern1?
No
Yes
Execute the commands in command-list1
Is test-string equal to pattern2?
No
Yes
Execute the commands in command-list2
...
Is test-string equal to patternN?
No
Yes
Execute the commands in command-listN
esac

FIGURE 12.8 Semantics of the `case` statement.

The `case` statement compares the value in **test-string** with the values of all the patterns one by one until either a match is found or no more patterns with which to match **test-string** remain. If a match is found, the commands in the corresponding **command-list** are executed and control goes out of the `case` statement. If no match is found, control goes out of `case`. However, in a typical use of the `case` statement, a wild card pattern matches any value of **test-string**. Also known as the *default case*, it allows the execution of a set of commands to handle an exception (i.e., error) condition for situations in which the value in **test-string** does not match any pattern. Back-to-back semicolons (`;;`) are used to delimit a **command-list**. Without `;;` the first command in the command list for the next pattern is executed, resulting in an unexpected behavior by the program. Figure 12.8 depicts the semantics of the `case` statement.

The following script in the **case_demo** file shows a simple but representative use of the `case` statement. It is a menu-driven program that displays a menu of options and prompts you to enter an option. Your option is read into a variable called `option`. The `case` statement then matches your option with one of the four available patterns (single characters in this case) one by one, and when a match is found, the corresponding **command-list** (a single command in this case) is executed. Thus, at the prompt, if you type `d` and hit `<Enter>`, the `date` command is executed and control goes out of `case`. Then, the program exits after the `exit 0` command executes. A few sample runs of the script follow the code in this session.

```
$ cat case_demo
#!/bin/sh
echo "Use one of the following options:"
echo "  d:    To display today's date and present time"
echo "  l:    To see the listing of files in your present working
              directory"
echo "  w:    To see who's logged in"
echo "  q:    To quit this program"
echo -n "Enter your option and hit <Enter>: "
read option
case "$option" in
        d)    date
                ;;
        l)    ls
                ;;
        w)    who
                ;;
        q)    exit 0
                ;;
esac
exit 0
$ ./case_demo
Use one of the following options:
  d:    To display today's date and present time
  l:    To see the listing of files in your present working
        directory
  w:    To see who's logged in
  q:    To quit this program
Enter your option and hit <Enter>: d
Sat Jul 26 17:28:19 PKT 2014
$ ./case_demo
Use one of the following options:
  d:    To display today's date and present time
  l:    To see the listing of files in your present working
        directory
  w:    To see who's logged in
  q:    To quit this program
Enter your option and hit <Enter>: w
malik           pts/0        Jul 15 08:11
davis           pts/1        Jul 27 16:11 (39.59.91.120)
sarwar          pts/2        Jul 27 16:48
jacob           pts/3        Jul 26 16:03
$ ./case_demo
Use one of the following options:
  d:    To display today's date and present time
  l:    To see the listing of files in your present working
        directory
  w:    To see who's logged in
  q:    To quit this program
Enter your option and hit <Enter>: a
$
```

From the output of the w option, it seems that **davis** is using the system through a remote login, most likely through an ssh session. Note that, when you enter a valid option, the expected output is displayed. However, when you enter input that is not a valid option (a in the preceding session), the program does not give you any feedback. The reason is that the case statement matches your input with all the patterns, one by one, and exits when there is no match. We need to modify the script slightly so that when you enter an invalid option, the script tells you so and then terminates. To do so we add the following code.

```
*)     echo "Invalid option; try running the program again."
       exit 1
       ;;
```

We also enhance the script so that uppercase and lowercase inputs are considered to be the same. We use the pipe symbol (|) in the patterns to specify a logical OR operation. The enhanced code and some sample runs are shown in the following session.

```
$ cat case_demo
echo "Use one of the following options:"
echo "  d or D:    To display today's date and present time"
echo "  l or L:    To see the listing of files in your present
working directory"
echo "  w or W:    To see who's logged in"
echo "  q or Q:    To quit this program"
echo -n "Enter your option and hit <Enter>: "
read option
case "$option" in
        d|D)    date
                ;;
        l|L)    ls
                ;;
        w|W)    who
                ;;
        q|Q)    exit 0
                ;;
        *)      echo "Invalid option; try running the program
                again."
                exit 1
                ;;
esac
exit 0
$ ./case_demo
Use one of the following options:
  d or D:    To display today's date and present time
  l or L:    To see the listing of files in your present working
             directory
  w or W:    To see who's logged in
  q or Q:    To quit this program
Enter your option and hit <Enter>: D
Sun Jul 27 16:49:47 PKT 2014
$ ./case_demo
Use one of the following options:
  d or D:    To display today's date and present time
  l or L:    To see the listing of files in your present working
             directory
  w or W:    To see who's logged in
  q or Q:    To quit this program
Enter your option and hit <Enter>: d
Sun Jul 27 16:50:18 PKT 2014
$ ./case_demo
Use one of the following options:
  d or D:    To display today's date and present time
  l or L:    To see the listing of files in your present working
             directory
  w or W:    To see who's logged in
  q or Q:    To quit this program
Enter your option and hit <Enter>: a
Invalid option; try running the program again.
$
```

12.7 COMMAND GROUPING

A number of shell commands may be run as a group. The following is a brief description of command grouping.

SYNTAX

(command-list)
Or
{ command-list; }

Purpose: Syntax 1: To execute commands in **command-list** under a child of the current Bourne shell
Syntax 2: To execute commands in **command-list** as a group

The child shell inherits a copy of the parent shell's environment, except trapped but ignored signals. When commands are executed by using the second syntax, no child shell is created. It allows redirection of the outputs of the commands in **command-list**. In the second syntax, it is mandatory to have a semicolon at the end of the last command, a space after {, and a space before }. The following interactive session shows a few examples of command grouping using both syntaxes.

```
$ { date; }
Sat Aug  2 06:34:51 PKT 2014
```

```
$ { date; echo -n "Hello, "; echo "world!"; }
Sat Aug  2 06:35:27 PKT 2014
Hello, world!
$ { date; echo -n "Hello, "; echo "world!"; } > groupout
$ more groupout
Sat Aug  2 06:35:40 PKT 2014
Hello, world!
$ { date; echo; ps; echo; echo "Hello, world!"; }
Sat Aug  2 06:36:39 PKT 2014
  PID TT  STAT    TIME COMMAND
31078  1  Is   0:00.21 -csh (csh)
31113  1  S    0:00.05 /bin/sh
31371  1  R+   0:00.01 ps
Hello, world!
$ (date; pwd)
Sun Aug  3 12:47:24 PKT 2014
/home/sarwar
$ (date; pwd) > groupout
$ cat groupout
Sun Aug  3 12:52:42 PKT 2014
/home/sarwar
$ (date; echo; ps; echo; echo "Hello, world!")
Sat Aug  2 06:37:18 PKT 2014
  PID TT  STAT    TIME COMMAND
31078  1  Is   0:00.21 -csh (csh)
31113  1  S    0:00.06 /bin/sh
31421  1  S+   0:00.00 /bin/sh
31423  1  R+   0:00.02 ps
Hello, world!
$
```

The outputs of the commands in this session are self-explanatory. Note that no child shell is created to execute the commands in `{ date; echo; ps; echo; echo "Hello, world!"; }`. However, the commands in `(date; echo; ps; echo; echo "Hello, world!")` are executed under a child shell and the PID of the child shell is **31421**.

SUMMARY

Every UNIX shell has a programming language that allows you to write programs for performing tasks that cannot be performed by existing commands. These programs are commonly known as shell scripts. In its simplest form, a shell script consists of a list of shell commands that are executed by a shell one by one, sequentially. More advanced scripts contain program control flow statements for implementing multiway branching and repetitive execution of a block of commands in a script. The shell programs that consist of Bourne shell commands, statements, and features are called Bourne shell scripts.

The shell variables are main memory locations that are given names and can be read from and written to. There are two types of shell variables: environment variables and user-defined variables. The environment variables are initialized by the shell at the time of user login and are maintained by the shell to provide a nice work environment. The user-defined variables are used as scratch pads in a script to accomplish the task at hand. Some environment variables such as positional parameters are read only in the sense that you cannot change their values without using the `set` command. User-defined variables can also be made read only by using the `readonly` command.

The Bourne shell commands for processing shell variables are = (for assigning a value to a variable), `set` (for setting values of positional parameters and displaying values of all environment variables), `env` (for displaying values of all shell variables), `export` (for allowing subsequent commands to access shell variables), `read` (for assigning values to variables from the keyboard), `readonly` (for making user-defined variables read only), `shift` (for shifting command line arguments to the left by one or more positions), `unset` (to reset the value of a read/write variable to null), and `test` (to evaluate an expression and return true or false).

The program control flow statements `if` and `case` allow the programmer of a shell script to implement multiway branching; the `for`, `until`, and `while` statements allow the programmer to implement loops; and the `break` and `continue` statements allow the user to interrupt sequential execution of a loop in a script. I/O redirection, command substitution, and other shell features can be used with control flow statements as with other shell commands (see Chapter 13).

QUESTIONS AND PROBLEMS

1. What is a shell script? Describe three ways of executing a shell script.
2. What is a shell variable? What is a read-only variable? How can you make a user-defined variable read only? Give an example to illustrate your answer.
3. Which shell environment variable is used to store your search path? Change your search path interactively to include **~/bin** and your current directory (.). What would this change allow you to do? Why? If you want to make it a permanent change, what would you do? See Chapter 4 if you have forgotten how to change the search path of your shell.
4. What will be the output if the shell script **read_demo** in Section 12.3.6 is executed and you give `*` as input each time you are prompted?
5. Write a shell script that takes an ordinary file as an argument and removes the file if its size is zero. Otherwise, the script displays the following information about the file: name, size (in bytes), number of hard links, owner, and modify date (in this order) on one line. Your script must do the appropriate error checking.

6. Write a shell script that takes a directory as a required argument and displays the names of all zero-length files in it. Do the appropriate error checking.
7. Write a shell script that removes all zero-length ordinary files from the current directory. Do appropriate error checking.
8. Modify the script in Problem 6 so that it removes all zero-length ordinary files in the directory passed as an optional argument. If you don't specify the directory argument, the script uses the present working directory as the default argument. Do the appropriate error checking.
9. Run the script in **if_demo2** in Section 12.6.1 with `if _ demo2` as its argument. Does the output make sense to you? Why or why not?
10. Write a shell script that takes a list of login names on your computer system as command line arguments and displays these login names and full names of the users who own these logins (as contained in the **/etc/passwd** file), one per line. If a login name is invalid (i.e., not found in the **/etc/passwd** file), display the login name but nothing for the full name. The format of the output line is `login name: username`.
11. What happens when you run a stand-alone command enclosed in back quotes (grave accents), such as `` `date` ``? Why?
12. What happens when you type the following sequence of `shell` commands?
    a. `name=date`
    b. `$name`
    c. `` `$name` ``
13. Look at your **~/.profile** and **/etc/profile** files and list the environment variables that are exported along with their values. What is the purpose of each variable?
14. Write a Bourne shell script that takes a list of login names as its arguments and displays the number of terminals that each user is logged on to in a LAN environment.
15. Write a Bourne shell script **domain2ip** that takes a list of domain names as command line arguments and displays their IP addresses. Use the `nslookup` command. The following is a sample run of this program.

```
$ domain2ip usc.edu up.edu redhat.com
Name: usc.edu
Address: 128.125.253.136
Name: up.edu
Address: 64.251.254.23
```

```
Name: redhat.com
Address: 209.132.183.105
$
```

16. Modify the script in the **case_demo** file in Section 12.6.6 so that it allows you to try any number of options and quits only when you use the `q` option.

17. Write a Bourne Shell script that displays the following menu and prompts you for one-character input to invoke a menu option, as follows:

    a. List all files in the present working directory

    b. Display today's date and time

    c. Invoke the shell script for Problem 14

    d. Display whether a file is a *simple* file or a *directory*

    e. Create a backup for a file

    f. Start a secure shell (ssh) session

    g. Start an ftp session

    h. Exit

    Option (c) requires that you ask for a list of login names; and for options (d) and (e), insert a prompt for file names before invoking a shell command/program. For options (f) and (g), insert a prompt for a domain name (or IP address) before initiating an ssh or ftp session. The program should allow you to try any option any number of times and should quit only when you give option `x` as input. A good programming practice for you to adopt is to build code incrementally—that is, write code for one option, test it, and then go to the next option.

18. Modify the Bourne Shell script for Problem 17 so that it executes code for each option under a child shell. Display the PID for the child shell whenever it is initiated to run the code for an option, before the code is executed.

19. What is the purpose of the `echo *` command? Run the command on your system and explain the output of the command.

20. The Bourne shell allows the following types of command groupings: `(command-list)` and `{ command-list; }`. Run the following commands on your system and answer the questions that follow. If there are any errors, correct them and describe those corrections.

    a. `(date; pwd; who; echo "Hello, \c"; echo "world!")`

    b. `{date; pwd; who; echo "Hello, world!"}`

Make appropriate changes in the corrected versions of the two command groups so that, in each case, the outputs of the `date` and `pwd` commands go to **file1** and outputs of the `who` and `echo` commands are redirected to **file2**. Then, demonstrate that commands in the first group are executed under a child shell and those in the second group are executed under the current shell. For the first case, show two ways of displaying the PID of the child shell.

CHAPTER 13

Advanced Bourne Shell Programming

Objectives

- To discuss numeric data processing
- To describe how standard input of a command in a shell script can be redirected from data within the script
- To explain the signal/interrupt processing capability of the Bourne shell
- To describe how file I/O can be performed by using file descriptors and how standard files can be redirected from within a shell script
- To explain functions in the Bourne shell
- To discuss debugging of Bourne shell scripts
- To cover the commands and primitives

  |, <, >, >>, `clear`, `exec`, `expr`, `grep`, `kill`, `more`, `read`, `sort`, `stty`, `trap`

13.1 INTRODUCTION

We discuss several important, advanced features of the Bourne shell in this chapter. They include processing of numeric data, the *here document*, signals and signal processing, and redirection of standard files from within a shell script. We also discuss the Bourne shell's support of functions that allow the programmer to write general-purpose and modular code. Finally, we describe how Bourne shell scripts can be debugged.

13.2 NUMERIC DATA PROCESSING

The values of all Bourne shell variables are stored as character strings. Although this feature makes symbolic data processing fun and easy, it does make numeric data processing a bit challenging. The reason is that integer data is actually stored in the form of character strings. In order to perform arithmetic and logic operations on them, you need to convert them to integers, perform the necessary operations on the integer data, and be sure the result is converted back to a character string for its proper storage in a shell variable. Fortunately, the Bourne shell command `expr` does the trick. The following is a brief description of the command.

SYNTAX

`expr args`

Purpose: Evaluate the expression arguments **`args`** and send the result to standard output
Commonly used options/features:

`|`	Force creation of link; don't prompt if **new-file** already exists
`\&`	Don't create the link if **new-file** already exists
`=, \>, \>=,` `\<, \<=, !=`	Integer comparison operators: equal, greater than, greater than or equal to, less than, less than or equal to, not equal
`+, -, \*, /, %`	Integer arithmetic operators: add, subtract, multiply, integer divide (return quotient), remainder

Shell metacharacters such as `*` must be escaped in an expression so that they are treated literally and not as shell metacharacters. In the following session, the first `expr` command increments the value of the shell variable `var1` by 1. The second `expr` command computes the square of `var1`. The last two `echo` commands show the use of the `expr` command to perform integer division and integer remainder operations on `var1`.

```
$ var1=10
$ var1=`expr $var1 + 1`
$ echo $var1
11
$ var1=`expr $var1 \* $var1`
$ echo $var1
121
$ echo `expr $var1 / 10`
12
$ echo `expr $var1 % 10`
1
$
```

The following `countup` script takes an integer as a command line argument and displays the range of numbers from 1 to the given number in one line, in ascending order. In the script, we use a simple `while` loop to display the current number (starting with 1) and then compute the next numbers, until the current number becomes greater than the number passed as the command line argument.

```
$ cat countup
#!/bin/sh
if [ $# != 1 ]
  then
       echo "Usage: $0 integer-argument"
       exit 1
fi
target="$1"     # Set target to the number passed at the command
                # line
current=1       # The first number to be displayed
# Loop here until the current number becomes greater than the target
while [ $current -le $target ]
do
        echo -n "$current"
        current=`expr $current + 1`
done
echo    # Move cursor to the next line
exit 0
$ ./countup 5
1 2 3 4 5
$
```

The following script, addall, takes a list of integers as command line arguments and displays their sum. The while loop adds the next number in the argument list to the running sum (which is initialized to 0), updates the count of numbers that have been added, and shifts the command line arguments left by one position. The loop then repeats until all the numbers in the command line arguments have been added. The sample run following the code takes the list of the first eight perfect squares and returns their sum.

```
$ cat addall
#!/bin/sh
# File Name: ~/unixbook/examples/Bshell/addall
# Author:    Syed Mansoor Sarwar
# Written:   August 18, 2004
# Modified:  August 18, 2004, July 28, 2014
# Purpose:   To demonstrate use of the expr command in processing
#            numeric data
# Brief Description:
#    Maintain the running sum of numbers in a numeric
#    variable called sum, initialized to 0. Read the next
#    integer and add it to sum. When all the integers
#    specified as command line arguments have been read,
#    display the answer, and terminate the program. If
#    the program is run with no arguments, inform the
#    user of the command syntax.
if [ $# = 0 ]
```

```
  then
     echo "Usage: $0 number-list"
     exit 1
fi
sum=0    # Running sum initialized to 0
count=0  # Count the count of numbers passed as arguments
while [ $# != 0 ]
do
  sum=`expr $sum + $1`      # Add the next number to the running sum
  count=`expr $count + 1`   # Update count of numbers added so far
  shift                     # Shift the counted number out
done
# Display final sum
echo "The sum of the given $count numbers is $sum."
exit 0
$ ./addall
Usage: ./addall number-list
$ ./addall 1 4 9 16 25 36 49 64
The sum of the given 8 numbers is 204.
$
```

Although this example neatly explains numeric data processing, it is nothing more than an integer addition machine. We now present a more useful example that uses the UNIX file system. The **fs** (for file size) file contains a script that takes a directory as an optional argument and returns the size (in bytes) of all nondirectory files in it. On some UNIX systems, running the `fs` command invokes `xfs`. If this happens on your system, change the name of this script to **files**, or whatever name you prefer to use.

When you run the program without a command line argument, the script assumes your current directory as the argument. If you run it with more than one command line argument, the script displays the command syntax and terminates. When you execute it with one nondirectory argument only, again the program displays the command syntax and exits. If the program is run with a nonexistent file as an argument, it displays an error message and terminates.

The gist of this script is the following code that runs when the script is run with a directory as a command line argument.

```
ls $directory | more | while read file
do
...
Done
```

This code generates a list of files in `directory` with the `ls` command, converts the list into one file name per line list with the `more` command, and reads each file name in the list, one by one, with the `read` command until no file remains in the list. The `read` command returns true if it reads a line and returns false when it reads the `eof` marker.

The body of the loop—that is, the code between do and done—adds the file size to the running total if the file is an ordinary file. When no name is left in the directory list, the program displays the total space in bytes occupied by all nondirectory files in the directory and terminates.

If the value of the `file` variable is not an existing file, the `[ ! -e "$file" ]` expression returns false and the error message `Usage: fs [directory name]`, as shown in sample run, where **unix3e** is a nonexistent directory. On some systems, the message may display `./fs` instead of `fs`, as in our session. The `file="$directory"/"$file"` statement is used to construct the relative pathname of a file with respect to the directory specified as the command line argument. Without this, the `set -- 'ls -l "$file"`` command will be successful only if the directory contains the name of the current directory.

```
$ cat fs
#!/bin/sh
# File Name:  ~/unix3e/BourneShell/fs
# Author:     Syed Mansoor Sarwar
# Written:    August 18, 2004
# Modified:   May 8, 2004, August 20, 2004, Jul 28, 2014
# Purpose:    To add the sizes of all the files in a directory
#             passed as command line argument
# Brief Description:
#    Maintain running sum of file sizes in a numeric variable
#    called sum, starting with 0. Read all the file names
#    by using the pipeline of the ls, more, and while commands.
#    Get the size of the next file and add it to the running
#    sum. Stop when all file names have been processed and
#    display the answer.
if [ $# = 0 ]    # If no command line argument, the
                 # set directory to current directory
   then
      directory="."
   elif [ $# != 1 ]     # If more then one command line argument
                        # then display command syntax
   then
      echo "Usage: $0 [directory name]"
      exit 1
   elif [ ! -e "$1" ]   # If one command line argument, but file
                        # does not exist, display error message
   then
      echo "$1: File does not exist"
      exit 1
   elif [ ! -d "$1" ]   # If one command line argument, but is
                        # not a directory, show command syntax
   then
      echo "Usage: $0 [directory name]"
      exit 1
   else
      directory="$1"    # If one command line argument and it is a
                        # directory, prepare to perform the task
fi
# Get file count in the given directory; for empty directory,
  display a
# message and quit.
file_count=`ls $directory | wc -w`  # Get count of files in the
                                    # directory
if [ $file_count -eq 0 ]            # If no files, display error
                                    # message
   then
      echo "$directory: Empty directory."
      exit 0
fi
# For each file in the directory specified, add the file size
# to the running total. The more command is used to output file
# names one per line so can read command can be used to read
# file names.
sum=0 # Running sum initialized to 0.
ls "$directory" | more |
while read file
do
   file="$directory"/"$file"  # Store the relative path name for
                              # each file
   if [ -f "$file" ]          # If it is an ordinary file
      then  # then
         set -- `ls -l "$file"` # Set command line arguments
         sum=`expr $sum + $5`   # Add file size to the running
                                # total.
   fi
   # Code to decrement the file_count variable and display the
   # final sum if the last file has been processed.
   if [ "$file_count" -gt 1 ] # Are more files left? If so,
                              # continue.
      then
         file_count=`expr $file_count - 1`
      else
      # Spell out the current directory
      if [ "$directory" = "." ]
         then
            directory="your current directory"
      fi
      echo "The size of all ordinary files in $directory is $sum
bytes."
```

```
   fi
done
exit 0
$
$ pwd
/home/sarwar/unix3e/ch13
$ ./fs / /bin
Usage: ./fs [directory name]
$ ./fs unix3e
unix3e: File does not exist
$ ./fs ~/unix3e
The size of all ordinary files in /home/sarwar/unix3e is 0 bytes.
$ ./fs ..
The size of all ordinary files in .. is 0 bytes.
$ ./fs .
The size of all ordinary files in your current directory is 4716
bytes.
$ ./fs /
The size of all ordinary files in / is 10238 bytes.
$ ./fs /bin
The size of all ordinary files in /bin is 1635367 bytes.
$ ./fs dir1
dir1: Empty directory.
$
```

In the following in-chapter exercise, you will create a Bourne shell script that processes numeric data by using the `expr` command.

EXERCISE 13.1

Create the **addall** script in your directory and run it with the first 10 numbers in the Fibonacci series. What is the result? Does the program produce the correct result? If you are not familiar with the Fibonacci series, browse through the following website: http://en.wikipedia.org/wiki/Fibonacci_number.

13.3 THE HERE DOCUMENT

The *here document* feature of the Bourne shell allows you to redirect standard input of a command in a script and attach it to data within the script, wrapped in a particular format, as will be explained. Obviously, this feature works with commands that take input from standard input. The feature is used mainly to display menus, although there are some other important uses of this feature. The advantage of maintaining data within the script is that it eliminates extra file operations such as `open` and `read` that would be required if the data was maintained in a separate file. The result is a much faster program. The following is a brief description of the here document.

SYNTAX

```
command << [-] input_marker
... input data ...
input_marker
```

Purpose: To execute `command` with its input coming from the here document—data between the input start and end markers `input _ marker`

The `input _ marker` is a string that you choose to wrap the input data in for `command`. The closing marker must be on a line by itself and cannot be surrounded by any spaces. The command and variable substitutions are performed before the here document data is directed to **stdin** of the command. Quotes can be used to prevent these substitutions or to enclose any quotes in the here document. `input _ marker` can be enclosed in quotes to prevent any substitutions in the entire document, as in:

```
command <<'Marker'
...
'Marker'
```

A hyphen (-) after << can be used to remove leading tabs (not spaces) from the lines in the here document and the marker that ends the here document. This feature allows the here document and the delimiting marker to conform to the indentation of the script. The following session illustrates this point:

```
while [ ... ] do
grep ... <<- DIRECTORY
            John Doe ...
            ...
            Art Pohm ... DIRECTORY
...
done
```

One last, but very important point: output and error redirections of the command that uses the here document must be specified in the command line, not following the marker that ends the here document. The same is true of connecting the standard output of the command with other commands via a pipeline, as shown in the following session. Note that the `grep ... <<- DIRECTORY 2> errorfile | sort` command can be replaced by `(grep ... 2> errorfile | sort) <<- DIRECTORY`.

```
while [ ... ] do
      grep ... <<- DIRECTORY 2> errorfile | sort John Doe ...
```

```
                    ...
        Art Pohm ... DIRECTORY
                    ...
Done
```

We can illustrate the use of the here document feature with a simple instance of redirecting **stdin** of the `cat` command from the here document. The script in the **heredoc_demo** file is used to display a message for the user and then send a message to the person whose e-mail address is passed as a command line argument. In the following session, we use two here documents: one begins with `<< DataTag` and ends with `DataTag`; and the other begins with `<< WRAPPER` and ends with `WRAPPER`.

```
$ cat heredoc_demo
#!/bin/sh
cat << DataTag
This is a simple use of the here document. This data is the
input to the cat command.
DataTag

# Second example
mail -s "Weekly Meeting Reminder" $1 << WRAPPER

Hello,

This is a reminder for the weekly faculty meeting tomorrow.

Mansoor

WRAPPER

echo "Sending mail to $1 ... done."
exit 0
$ ./heredoc_demo eecsfaculty
This is a simple use of the here document. These data are the
input to the cat command.
Sending mail to eecsfaculty ... done.
$
```

The following script is more useful and makes a better utilization of the here document feature. The `dext` (directory expert) script maintains a directory of names, phone numbers, and e-mail addresses. The script is run with a name as a command line argument and uses the `grep` command to display the directory entry corresponding to the name. The `-i` option is used with the `grep` command in order to ignore the case of letters.

```
$ more dext
#!/bin/sh
if [ $# = 0 ]
   then
        echo "Usage: $0 name"
        exit 1
```

```
fi
user_input="$1"
grep -i "$user_input" << DIRECTORY
      John Doe        555.232.0000    johnd@somedomain.com
      Jenny Great     444.6565.1111   jg@new.somecollege.edu
      David Nice      999.111.3333    david_nice@xyz.org
      Don Carr        555.111.3333    dcarr@old.hoggie.edu
      Masood Shah     666.010.9820    shah@Garments.com.pk
      Jim Davis       777.000.9999    davis@great.adviser.edu
      Art Pohm        333.000.8888    art.pohm@great.professor.edu
      David Carr      777.999.2222    dcarr@net.net.gov
DIRECTORY
exit 0
$ ./dext
Usage: ./dext name
$ ./dext Pohm
     Art Pohm     333.000.8888  art.pohm@great.professor.edu
$ ./dext Carr
     Don Carr     555.111.3333  dcarr@old.hoggie.edu
     David Carr   777.999.2222  dcarr@net.net.gov
$
```

If there are multiple entries for a name, the `grep` command displays all the entries. You can display the entries in sorted order by piping the output of the `grep` command to the `sort` command and enclosing them in parentheses, as in `(grep -i "$user_input" | sort)`. We enhance the `dext` script in Section 13.6 to include this feature, as well as take multiple names from the command line.

The following in-chapter exercise is designed to reinforce your understanding of the here document feature of the Bourne shell.

EXERCISE 13.2

Create the `dext` script on your system and run it. Try it with as many different inputs as you can think of. Does the script work correctly?

13.4 INTERRUPT (SIGNAL) PROCESSING

We discussed the basic concept of signals in Chapter 10, where we defined them as software interrupts that can be sent to a process. We also stated that the process receiving a signal can take one of the three possible actions:

1. Take the default action as defined by the UNIX kernel
2. Ignore the signal
3. Take a programmer-defined action

TABLE 13.1 Some Important Signals, Their Numbers, and Their Purpose

Signal Name	Signal #	Purpose
`SIGHUP` (hang up)	1	Informs the process when the user who ran the process logs out, and the process terminates
`SIGINT` (keyboard interrupt)	2	Informs the process when the user presses `<Ctrl+C>` and the process terminates
`SIGQUIT` (quit signal)	3	Informs the process when the user presses `<Ctrl+|>` or `<Ctrl+\>` and the process terminates
`SIGKILL` (sure kill)	9	Terminates the process when the user sends this signal to it with the `kill -9` command
`SIGSEGV` (segmentation violation)	11	Terminates the process upon memory fault when a process tries to access memory space that does not belong to it
`SIGTERM` (software termination)	15	Terminates the process when the `kill` command is used without any signal number
`SIGTSTP` (suspend/stop signal)	18	Suspends the process; usually `<Ctrl+Z>`
`SIGCHLD` (child finishes execution)	20	Informs the process of termination of one of its children

In UNIX, several types of signals can be sent to a running program. Some of these signals can be sent via hardware devices such as the keyboard, but all can be sent via the `kill` command, as discussed in Chapter 10. The most common event that causes a hardware interrupt (and a signal) is generated when you press `<Ctrl+C>` and is known as the keyboard interrupt. The default kernel-defined action of this event is that the foreground process terminates. Other events that cause a process to receive a signal include termination of a child process, a process accessing a main memory location that is not part of its address space, and a software termination signal caused by execution of the `kill` command without any signal number. The address space of a process is the main memory area that the process owns legally and is allowed to access. Table 13.1 presents a list of some important signals, their numbers, and their purpose. The signal numbers, or the corresponding symbolic names, can be used to generate the respective signals with the `kill` command.

The interrupt processing feature of the Bourne shell allows you to write programs that can ignore signals, take actions as defined by the UNIX kernel for those signals, or execute a specific sequences of commands when signals of particular types are sent to them. This feature is much more powerful than that of the C shell, which allows programs to ignore a keyboard interrupt (`<Ctrl+C>`) only (see Chapter 15). The `trap` command can be used to intercept signals. The following is a brief description of the command.

SYNTAX

```
trap ['command-list'] [signal-list]
```

Purpose: To intercept signals specified in **signal-list** and take default kernel-defined action, ignore the signals, or execute the commands in **command-list**; note that quotes around **command-list** are required

When you use the `trap` command in a script without any argument (i.e., no **command-list** and no **signal-list**), the script takes default actions when it receives signals.

Thus, using the trap command without any argument is redundant. When the trap command is used without any commands in single quotes, the script ignores the signals in **signal-list**. When both a **command-list** and a **signal-list** are specified, the commands in **command-list** execute when a signal specified in **signal-list** is received by the script.

Next, we enhance the script in the **while_demo** file from Chapter 12 so that you cannot terminate execution of this program with <Ctrl+C> (signal number 2), the kill command without any argument (signal number 15), or the kill -1 command (to generate the SIGHUP signal). The enhanced version is in the **trap_demo** file, as shown in the following session. Note that the trap command is used to ignore signals 1, 2, 3, 15, and 18. A sample run illustrates this point.

```
$ cat trap_demo
#!/bin/sh

# Intercept signals 1, 2, 3, 15, and 18 and ignore them
trap '' 1 2 3 15 18

# Set the secret code
secretcode=agent007

# Get user input
echo "Guess the code!"
echo -e "Enter your guess: c"
read yourguess

# As long as the user input is the secret code (agent007 in this
# case), loop here: display a message and take user input again.
# When the user input matches the secret code, terminate the loop
# and execute the echo command that follows.
while [ "$secretcode" != "$yourguess" ]
do
  echo "Good guess but wrong. Try again!"
  echo -e "Enter your guess: \c"
  read yourguess
done
echo "Wow! You are a genius!"
exit 0
$ ./trap_demo
Guess the code!
Enter your guess: codecracker
Good guess but wrong. Try again!
Enter your guess: <Ctrl+C>
Good guess but wrong. Try again!
Enter your guess: agent007
Wow! You are a genius!
$
```

To terminate programs that ignore terminal interrupts, you have to use the `kill` command. You can do so by suspending the process by pressing <`Ctrl+Z`>, using the `ps` command to get the PID of the process, and terminating it with the `kill` command. Alternatively, you can login from another terminal and use the `ps` and `kill` commands as stated in the previous sentence.

You can modify the script in the **trap_demo** file so that it ignores signals 1, 2, 3, 15, and 18, clears the display screen, and turns off the echo. When echo has been turned off, whatever input you enter from the keyboard, is not displayed. Next, it prompts you for the code word and saves it. It then prompts you again to enter the same code word in order to make sure that you remember the word that you have entered. It gives you two chances for this purpose. If you do not enter the same code word both times, it reminds you of your bad short-term memory and quits. If you enter the same code word, it clears the display screen and prompts you to guess the code word again. If you do not enter the original code word, the display screen is cleared and you are prompted to guess again. The program does not terminate until you have entered the original code word. When you do enter it, the display screen is cleared, a message is displayed at the top left of the screen, and the echo is turned on. Because the terminal interrupt is ignored, you cannot terminate the program by pressing <`Ctrl+C`>. The `stty -echo` command turns off the echo. Thus, when you type the original code word (or any guesses), it is not displayed on the screen. The `stty echo` turns on the echo. The `clear` command clears the display screen and positions the cursor at the top-left corner. The resulting script is in the **canleave** file, as shown in the following session.

```
$ cat canleave
#!/bin/sh
# File Name:   ~/unix3e/BourneShell/canleave
# Author:      Syed Mansoor Sarwar
# Written:     August 18, 2004
# Modified:    May 8, 2004, Jul 29, 2014
# Purpose:     To allow a user to leave his/her terminal for a
#              short duration of time by locking the terminal
#              after taking a code from the user. Terminal is
#              unlocked only when the user re-enters the same
#              code. Ignores command line arguments.
# Brief Description:
#     Clear screen and turn off echo (i.e., do not display what
#     the user types at the keyboard). Take user code, save it,
#     and ask the user to re-enter his/her code just to make sure
#     that the user remembers the code that he/she has entered.
#     It is done twice. If the user does not enter the same code,
#     the program terminates after displaying a message for the
#     user. The user is prompted to enter the original code. If
#     the user enters the wrong code, the program keeps on
#     prompting the user until he/she enters the original code.
#     The keyboard is then unlocked, echo is turned on, and
#     program exits.
# Ignore signals 1, 2, 3, 15, and 18
trap '' 1 2 3 15 18

# Clear the screen, locate the cursor at the top-left corner,
# and turn off echo
clear
stty -echo

# Set the secret code
echo -n "Enter your code word: "
read secretcode
echo " "

# To make sure that the user remembers the code word,
# ask the user to enter the secrete code again.
echo -n "Enter your code word again: "
read same
if [ $secretcode != $same ]
  then
    clear
    echo "Wrong code. Try again."
fi
echo -n "Enter your code word again: "
read same
if [ $secretcode != $same ]
  then
    clear
    echo "Work on your short-term memory before using this code!"
    echo "Goodbye!"
    exit 1
fi

# Keyboard locked. Hit <Enter> to continue.
clear
echo -n "Keyboard locked. Hit <Enter> to continue."
read ignore
clear

# Get user guess to unlock the terminal
clear
echo -n "Enter the code word: "
read yourguess
echo " "

# As long as the user input is not the original code word, loop
# here: display a message and take user input again. When the user
# input matches the secret code word, terminate the loop and
# execute the following echo command.
while [ "$secretcode" != "$yourguess" ]
do
```

```
   clear
   echo "Wrong code. Try again."
   echo -n "Enter the code word: "
   read yourguess
done
# Set terminal to echo mode clear
clear
echo "Back again!"
stty echo
exit 0
$
```

You can use this script to lock your terminal before you leave it for a short period of time—for example, to pick up a printout or get a can of soda; hence, the name **canleave** (can leave). Using it saves you the time otherwise required for the logout and login procedures.

The following in-chapter exercise is designed to reinforce your understanding of the signal-handling feature of a Bourne shell.

EXERCISES 13.3

Test the scripts in the **trap_demo** and **canleave** files on your UNIX system. Do they work as expected? Be sure that you understand them.

13.5 THE `exec` COMMAND AND FILE I/O

The `exec` command is the command-level version of the UNIX loader. Normally, the `exec` command is used to replace the current process with a new process. Thus, when executed under a shell, `exec cmd` overwrites the current shell with `cmd`. The `exec` command may also be used to open and close *file descriptors*.

When the `exec` command is used in conjunction with the redirection operators, it allows commands and shell scripts to read/write any type of files, including devices. In this section, we describe both uses of this command but focus primarily on the second use.

13.5.1 Execution of a Command (or Script) in Place of Its Parent Process

The `exec` command can be used to run a command (or a script) instead of the process, usually the shell, that executes this command. It works with all shells. The following is a brief description of the command.

SYNTAX

`exec command`

Purpose: Overwrite the code for **`command`** on top of the process that executes the **`exec`** command (the calling process), which makes **`command`** run in place of the calling process without creating a new process

After you have run this command, the control cannot return to the calling process. When the `exec` command has finished, control goes back to the parent of the calling process. If the calling process is your login shell, control goes back to the `getty` process, which displays the `Login:` prompt after the `exec` command finishes execution, as in:

```
% exec date
Thu Jul 31 17:57:39 PKT 2014
Login:
```

When `exec date` finishes, control does not go back to the shell process but to the `getty` process—that is, the parent of the login shell process. The semantics of this command execution are shown in Figure 13.1.

If the command is run under a subshell of the login shell, control goes back to the login shell, as clarified in the following session. Here, a C shell is run as a child of the login shell, also a C shell in this case, and `exec date` is run under the child C shell. When the `exec date` command finishes execution, control goes back to the login C shell. The sequence of three diagrams from left to right shown in Figure 13.2 depicts the semantics of these steps.

```
% ps
  PID TT  STAT    TIME COMMAND
29301  4  Ss   0:00.05 -csh (csh)
41055  4  R+   0:00.00 ps
% /bin/csh
% ps
  PID TT  STAT    TIME COMMAND
29301  4  Ss   0:00.06 -csh (csh)
41084  4  S    0:00.00 /bin/csh
41097  4  R+   0:00.01 ps
% exec date
Thu Jul 31 18:16:01 PKT 2014
% ps
  PID TT  STAT    TIME COMMAND
29301  4  Ss   0:00.06 -csh (csh)
41133  4  R+   0:00.01 ps
%
```

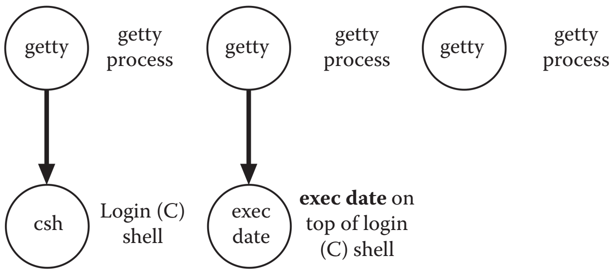


FIGURE 13.1 Execution of the `exec date` command under the login shell.

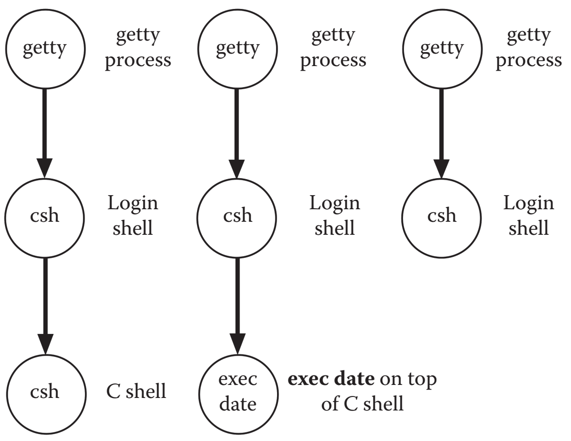


FIGURE 13.2 Execution of the `exec date` command under a subshell of the login shell.

13.5.2 File I/O via the `exec` Command

Different shells allow the use of different numbers of file descriptors at a time. As stated in Chapter 7, three of these descriptors are set aside for standard input (0), standard output (1), and standard error (2). Using the redirection operators with the `exec` command can use all of these descriptors for I/O. Table 13.2 describes the syntax of the `exec` command for file I/O.

When executed from the command line, the `exec < sample` command causes each line in the **sample** file to be treated as a command and executed by the current shell. That happens because the shell process, whose only purpose is to read commands from **stdin** and execute them, executes the `exec` command; as the **sample** file is attached to **stdin**, the shell reads its commands from this file. The shell terminates after executing the last

TABLE 13.2 Syntax of the `exec` Command for File I/O

Syntax	**Meaning**
`exec < file`	Opens `file` for reading and attaches standard input of the process to `file`
`exec > file`	Opens `file` for writing and attaches standard output of the process to `file`
`exec >> file`	Opens `file` for writing, attaches standard output of the process to `file`, and appends standard output to `file`
`exec n< file`	Opens `file` for reading and assigns it the file descriptor `n`
`exec n> file`	Opens `file` for writing and assigns it the file descriptor `n`

`exec n<< tag`
`...`
`tag`
	Opens a *here document* (data between `<< tag` and `tag`) for reading; the opened file is assigned a descriptor `n`

`exec n>> file`	Opens `file` for writing, assigns it file descriptor `n`, and appends data to the end of `file`
`exec n>&m`	Duplicates `m` into `n`; whatever goes into `file` with file descriptor `n` will also go into `file` with file descriptor `m`
`exec <&-`	Closes standard output
`exec >&-`	Closes standard output
`exec n<&-`	Closes `file` with descriptor `n` attached to **stdin**
`exec n >&-`	Closes `file` with descriptor `n` attached to **stdout**

line in **sample**. When executed from within a shell script, this command causes the **stdin** of the remainder of the script to be attached to **sample**. The following session illustrates the semantics of this command when it is executed at the command line. As shown, the **sample** file contains two commands: `date` and `echo`. A Bourne shell is run under the login shell, which is a C shell, via the `/bin/sh` command. When the `exec <  sample` command is executed, the commands in the **sample** file are executed, the Bourne shell (the child process of the login C shell) terminates after finishing execution of the last command in **sample** (the output of the third `ps` command shows that only the login shell runs after the `exec <  sample` command has completed execution), and control returns to the login shell.

```
% cat sample
date
echo "Hello, world!"
% ps
  PID TT  STAT    TIME COMMAND
41407  1  Ss    0:00.72 -csh (csh)
43286  1  R+    0:00.02 ps
% /bin/sh
% ps
  PID TT  STAT    TIME COMMAND
41407  1  Is    0:00.15 -csh (csh)
41802  1  S     0:00.04 /bin/sh
41901  1  R+    0:00.02 ps
% exec < sample
Thu Jul 31 18:41:38 PKT 2014
Hello, world!
% ps
  PID TT  STAT    TIME COMMAND
41407  1  Ss    0:00.50 -csh (csh)
41914  1  R+    0:00.00 ps
%
```

So, effectively, when the `exec <  sample` command is executed from the command line, it attaches **stdin** of the current shell to the sample file. When this command is executed from a shell script, it attaches **stdin** of the shell script to the sample file. In either case, the `exec <  /dev/tty` command must be executed to reattach **stdin** to the terminal. Here, **/dev/tty** is the pseudo terminal that represents the terminal on which the shell is executed. The following session illustrates the use of this command from the command line. The semantics of these steps are shown in Figure 13.3.

Similarly, when the `exec >  data` command is executed from the command line, it attaches **stdout** of the current shell to the data file. Thus, it causes outputs of all subsequent commands executed under the shell to go to the data file. Thus, you do not see the output of any command on the screen. In order to see the output on the screen again, you need to execute the `exec >  /dev/tty` command. After doing so, you can view the contents

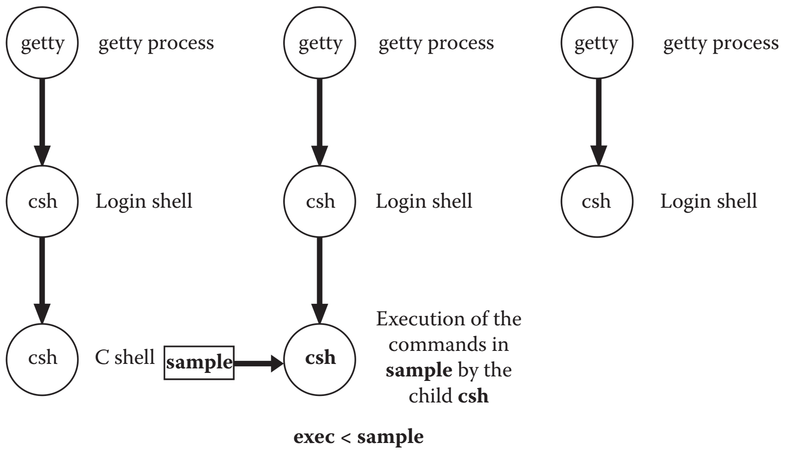


FIGURE 13.3 Execution of the `csh` and `exec < sample` commands under the login shell.

of the data file to see the outputs of all the commands executed prior to this command. When the `exec > data` command is executed from a shell script, it causes the outputs of all subsequent commands to go to the data file until the `exec > /dev/tty` command is executed from within the shell script.

The following session illustrates the use of this command from the command line. Note that, after the `exec > data` command has completed its execution, the outputs of all subsequent commands (`date`, `echo`, and `more`) go to the data file. In order to redirect the output of commands to the screen, the `exec > /dev/tty` command must be executed, as shown in the following session. Note that our system has an AMD64 CPU and, as expected, is running FreeBSD UNIX.

```
$ exec > data
$ date
$ echo "Hello, world!"
$ uname -sm
$ exec > /dev/tty
$ date
Thu Jul 31 19:27:46 PKT 2014
$ more data
Thu Jul 31 19:27:12 PKT 2014
Hello, world!
FreeBSD amd64
$
```

Similarly, you can redirect standard output and standard error for a segment of a shell script by using the following command:

```
exec > outfile 2> errorfile
```

In this case, output and error messages from the shell script following this line are directed to **outfile** and **errorfile**, respectively. (Obviously, file descriptor 1 can be used with **>** to redirect output.) If output needs to be reattached to the terminal, you can do so by using:

```
exec > /dev/tty
```

Once this command has executed, all subsequent output goes to the monitor screen. Similarly, you can use the `exec 2> /dev/tty` command to send errors back to the display screen.

Consider the following shell session. When `exec.demo1` is executed, **file1** gets the line containing `Hello, world!`, **file3** gets the contents of **file2**, and **file4** gets the line `This is great!`. The shell script between the commands `exec < file2` and `exec < /dev/tty` takes its input from **file2**. Therefore, the command `cat > file3` is really `cat < file2 > file3`. The `cat > file4` command takes input from the keyboard as it is executed after the `exec < /dev/tty` command has been executed (which reattaches the **stdin** of the script to the keyboard). Figure 13.4 illustrates the semantics of the three `cat` commands in the shell script.

```
$ more exec.demo1
cat > file1
exec < file2
cat > file3
exec < /dev/tty
cat > file4
$ ./exec.demo1
Hello, world!
<Ctrl+D>
This is great!
<Ctrl+D>
$ more file1
Hello, world!
$ more file2
[Contents of file2.]
$ more file3
[Contents of file3.]
$ more file4
This is great!
$
```

Now, we develop a shell script, `diff2`, which uses the file I/O features of the Bourne shell. It takes two files as command line arguments and compares them line by line. If the files are the same, it displays a message that says so, and the program terminates. If one file is smaller than the other, it displays a message informing you of that and exits. As soon as the program finds the lines at which the two files differ, it displays an error message informing you of the lines from both files that are different and terminates. The following is the script and a few sample runs.

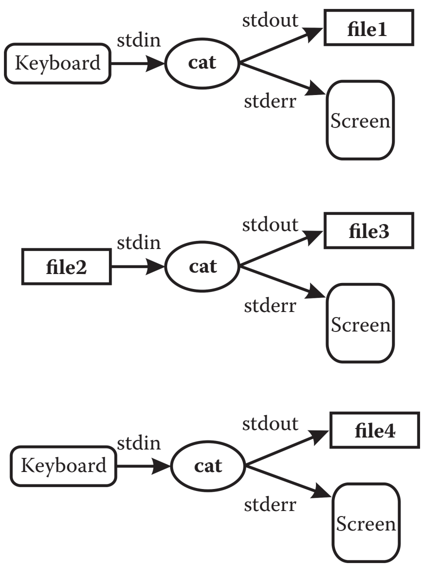


FIGURE 13.4 Detachment and reattachment of **stdin** and **stdout** inside a shell script.

```
$ cat diff2
#!/bin/sh

# File Name:  ~/unix3e/BourneShell/diff2
# Author:     Syed Mansoor Sarwar
# Written:    August 28, 2004
# Modified:   August 28, 2004, July 29, 2014
# Purpose: To see if the two files passed as command line
#     arguments are same or different
# Brief Description:
#     Read a line from each file and compare them. If
#     the lines are the same, continue. If they are
#     different, display the two lines and exit. If one
#     of the files finishes before the other, display a
#     message and exit. Otherwise, the files are the
#     same; display an appropriate message and exit

if [ $# != 2 ]
   then
      echo "Usage: $0 file1 file2"
      exit 1
   elif [ ! -f "$1" ]
   then
      echo "$1 is not an ordinary file"
      exit 1
   elif [ ! -f "$2" ]
   then
      echo "$2 is not an ordinary file"
      exit 1
   else
      :
fi
file1="$1"
file2="$2"

# Open files for reading and assign them file descriptors 3 and 4

exec 3< "$file1"
exec 4< "$file2"

# Read a line each from both files and compare. If both reach EOF,
# then files are the same. Otherwise, they are different. 0<&3 is
# used to attach standard input of the read line1 command to file
# descriptor 3, 0<&4 is used to attach standard input of the read
# line2 command to file descriptor 4.

while read line1 0<&3
do
   if read line2 0<&4
      then
      # if lines are different, the two files are not the same
         if [ "$line1" != "$line2" ]
            then
               echo "$1 and $2 are different."
               echo " $1: $line1"
               echo " $2: $line2"
               exit 1
         fi
      else
         # if EOF for file2 reached, file1 is bigger than file2
         echo "$1 and $2 are different and $1 is bigger than $2."
         exit 1
      fi
done
# if EOF for file1 reached, file2 is bigger than file1. Otherwise,
# the two files are the same. 0<&4 is used to attach standard
# input of read to file descriptor 4
if read line2 0<&4
   then
      echo "$1 and $2 are different and $2 is bigger than $1."
      exit 1
   else
      echo "$1 and $2 are the same!"
      exit 0
fi
# Close files corresponding to descriptors 3 and 4
```

```
exec 3<&-
exec 4<&-
$ cat test1
Hello, world!
Not the same!
Another line.
$ cat test2
Hello, world!
$ cat test3
Hello, world!
Not the same!
$ cat test4
This is different file.
Hello, world!
$ ./diff2
Usage: ./diff2 file1 file2
$ ./diff2 test1 test2 test3
Usage: ./diff2 file1 file2
$ ./diff2 test1 test1
test1 and test1 are the same!
$ ./diff2 test1 test3
test1 and test3 are different and test1 is bigger than test3.
$ ./diff2 test1 test2
test1 and test2 are different and test1 is bigger than test2.
$ ./diff2 test2 test3
test2 and test3 are different and test3 is bigger than test2.
$ ./diff2 test1 test4
test1 and test4 are different.
 test1: Hello, world!
 test4: This is different file.
$
```

The `exec` command is used to open and close files. The `exec 3< "$file1"` and `exec 4< "$file2"` commands open the files passed as command line arguments for reading and assigns them file descriptors 3 and 4. From this point on, you can read the two files by using these descriptors. The commands `read line1 0<&3` and `read line2 0<&4` read the next lines from the files with files descriptors 3 (for **file1**) and 4 (for **file2**), respectively. The commands `exec 3<&-` and `exec 4<&-` close the two files. The colon sign (:) in the `else` part of the first `if` statement is a null statement that simply returns true. You may use the Bourne shell command `true` for the same purpose. Incidentally, the `false` commands, as one would expect, returns false.

In the following in-chapter exercises, you will use the `exec` command to redirect the I/O of your shell to ordinary files. The concept of I/O redirection from within a shell script and file I/O by using file descriptors is also reinforced.

EXERCISE 13.4

Write a command for changing **stdin** of your shell to a file called **data** and **stdout** to a file called **out**, both in your present working directory. If the data file contains the following lines, what happens after the commands are executed?

```
echo -n "The time now is: "
date
echo -n "The users presently logged on are: "
who
```

EXERCISE 13.5

After finishing the steps in Exercise 13.4, what happens when you type commands at the shell prompt? Does the result make sense to you? Write the command needed to bring your environment back to normal.

EXERCISE 13.6

Create a file that contains the `diff2` script and try it with different inputs.

13.6 FUNCTIONS IN THE BOURNE SHELL

The Bourne shell allows you to write functions. Functions consist of a series of commands, called the *function body*, that are given a name. You can invoke the commands in the function body by using the function name.

13.6.1 Reasons for Using Functions

Functions are normally used if a piece of code is repeated at various places in a script. By making a function of this code, you save typing time. Thus, if a block of code is used at, say, nine different places in a script, you can create a function of it and invoke it where it is to be inserted by using the name of the function. Another advantage of functions is that any changes to the code, which would otherwise be needed at nine places, would now be needed at one place only—that is, in the function body. The trade-off is that the mechanism of transferring control to the function code and returning it to the calling code (from where the function is invoked/called) takes time, which slightly increases the running time of the script.

Another way of saving typing time is to create another script file for the block of code and invoke this code by calling the script as a command. The disadvantage of using this technique is that, as the script file is on a secondary storage device such as a hard disk, the invocation of the script requires loading the script from the disk into main memory once it has been defined, which is an expensive operation. Whether they are located in **~/.profile**, defined interactively in a shell, or defined in the script, function definitions are always in the main memory. Thus, invocation of functions is several times faster than invoking shell scripts, which are on the disk.

13.6.2 Function Definition

Before you can use a function, you have to define it. For often-used functions, you should put their definitions in your **~/.profile** file. This way, the shell records them in its environment when you log on and allows you to invoke them while you use the system. The definitions for functions that are specific to a script are usually put in the file that contains the script. You must execute the **~/.profile** file with the `.` (dot) command after defining a function in it and before using it, unless you want to log off and then log back on. You can also define functions while interactively using the shell. These definitions are valid for as long as you remain in the session that you were in when you defined these functions.

The syntax of a function definition is as given as follows:

```
name () command
```

where `name` is the name (also called the label) of the function and `command` is the body of the function. For a function with multiple commands in the body, the command list is enclosed in curly brace, { and }. This allows you to format a function definition in the following ways:

```
function_name () command

function_name () { command-list; }

function_name () {
  command-list
}

function_name ( )
{
  command-list
}
```

The `function _ name` is the name of the function that you choose, and the commands in **command-list** comprise the function body. The following session shows example function definitions corresponding to the four ways of defining functions and sample runs of the first three functions. The output of functions `function4` and `function5` would be the same as that of `function3`.

```
$ function1 () date
$ function2 () { uname -s; }
$ function3 () { date; echo "Hello, world!"; ps; }
$ function4 () {
> date
> echo "Hello, world!"
> ps
> }
$ function5 ()
```

```
> {
>     date
>     echo "Hello, world!"
>     ps
> }
$ function1
Sat Aug 2 18:11:32 PKT 2014
$ function2
FreeBSD
$ function3
Sat Aug 2 18:13:56 PKT 2014
Hello, world!
  PID TT  STAT    TIME COMMAND
82778  1  Is   0:00.17 -csh (csh)
82807  1  S    0:00.13 /bin/sh
85195  1  R+   0:00.02 ps
$
```

Note that there is no `export` command for functions. Thus, function definitions are visible only to the shell in which they are defined.

13.6.3 Function Invocation/Call

The commands in a function body are not executed until the function is invoked—that is, called. You can invoke a function by using its name as a command. When you call a function, its body is executed and control comes back to the command following the function call. If you invoke a function at the command line, control returns to the shell after the function finishes its execution.

Variables declared within a function are visible outside the function and may be accessed accordingly. In order to make a variable *local* to a function, the keyword `local` may be used, as in the function `f1` in the following example. A local variable inherits the value and *exported* and *read-only* flags of a variable with the same name defined in the surrounding scope. If there is no such variable in the surrounding scope, the local variable is initialized to null. The Bourne shell uses *dynamic scoping*. This means that a local variable defined in the caller function is accessible to the called function. It is also demonstrated in the following session, where the variable `x` is defined as local within `f1` (the caller function) and is accessible to `f2` (the called function). However, `x` is not accessible outside `f1`. On the other hand, the variable `y` is not defined as local and is accessible outside `f1`. The outputs of the `echo $x`, `echo $y`, and `f2` commands verify these scoping rules.

```
$ f1 () { local x=10; y=20; f2; }
$ f2 () { echo "x is $x"; echo "y is $y"; }
$ f1
x is 10
y is 20
$ echo $x
$ echo $y
20
$ f2
x is
y is 20
$
```

13.6.4 A Few More Examples of Functions

The execution of the following function, called `machines`, returns the names of all the computers on your local network.

```
$ machines ()
> {
>    date
>    echo "These are the machines on the network: "
>    ruptime | cut -f1 -d' ' | more
> }
$ ./machines
Thu Jul 31 20:23:28 PKT 2014
These are the machines on the network: upibm0
...
upsun1
...
upsun29
$
```

We now enhance the `dext` script described in Section 13.3 so that it can take multiple names at the command line. The enhanced version also uses a function `OutputData` to display one or more output records (i.e., lines in the directory) for every name passed as the command line argument. In the case of multiple lines for a name, this function displays them in sorted order. A few sample runs are shown following the script.

```
$ cat dext
#!/bin/sh
if [ $# = 0 ]
   then
      echo "Usage: $0 name"
      exit 1
fi
OutputData()
{
  echo "Infomation about $user_input"
  (grep -i "$user_input"| sort) << DIRECTORY
```

```
      John Doe    555.232.0000      johnd@somedomain.com
      Jenny Great 444.6565.1111     jg@new.somecollege.edu
      David Nice  999.111.3333      david_nice@xyz.org
      Don Carr    555.111.3333      dcarr@old.hoggie.edu
      Masood Shah 666.010.9820      shah@Garments.com.pk
      Jim Davis   777.000.9999      davis@great.adviser.edu
      Art Pohm    333.000.8888      art.pohm@great.professor.edu
      David Carr  777.999.2222      dcarr@net.net.gov

DIRECTORY
  echo      # A blank line between two records
}
# As long as there is at least one command line argument (name),
# take the first name, call the OutputData function to search the
# DIRECTORY and display the line(s) containing the name, shift
# this name left by one position, and repeat the process.

while [ $# != 0 ]
do
  user_input="$1" # Get the next command line argument (name)
  OutputData      # Display info about the next name
  shift           # Get the following name
done
exit 0
$ ./dext john
Infomation about john
      John Doe    555.232.0000  johnd@somedomain.com

$ ./dext jim
Infomation about jim
      Jim Davis   777.000.9999  davis@great.adviser.edu

$ ./dext pohm masood carr
Infomation about pohm
      Art Pohm    333.000.8888  art.pohm@great.professor.edu

Infomation about masood
      Masood Shah 666.010.9820  shah@Garments.com.pk

Infomation about carr
      David Carr  777.999.2222  dcarr@net.net.gov
      Don Carr    555.111.3333  dcarr@old.hoggie.edu

$
```

In the following in-chapter exercise, you will write a simple function and a Bourne shell script that uses it.

EXERCISES 13.7

Write a function called menu that displays the following menu. Then write a shell script that uses this function.

```
Select an item from the following menu:
d. to display today's date and current time,
f. to start an ftp session,
t. to start a telnet session, and
q. to quit.
```

13.7 DEBUGGING SHELL PROGRAMS

You can debug your Bourne shell scripts by using the -x (echo) option of the sh command. This option displays each line of the script after variable substitution but before its execution. You can combine the -x option with the -v (verbose) option to display each line of the script, as it appears in the script file, before execution. You can also invoke the sh command from the command line to run the script, or you can make it part of the script, as in #!/bin/sh -xv. In the latter case, remove the -xv options after debugging is complete.

In the following session, we show how a shell script can be debugged. The script in the **debug_demo** file prompts you for a digit. If you enter a value between 1 and 9, it displays Good input! and quits. If you enter any other value, it simply exits. When the script is executed and you enter 4, it displays the message debug _ demo: [4: not found.

```
$ cat debug_demo
#!/bin/sh

echo -e "Enter a digit: c"
read var1
if ["$var1" -ge 1 -a "$var1" -le 9 ]
   then
      echo "Good input!"
fi
exit 0
$ ./debug_demo
Enter a digit: 4
./debug_demo: [4: not found
$
```

We debug the program by using the sh -xv debug _ demo command. The shaded portion of the run-time trace shows the problem area. In this case, the error is generated because of a problem in the condition for the if statement. A closer examination of the shaded area reveals that a missing space between [and 4 is the problem. In other words, the comparison between [4 and 1 is the problem; it should be between 4 and 1. After we take care of this problem by changing ["$var1" to ["$var1", the script works properly.

```
$ sh -xv debug_demo
#!/bin/sh

echo -e "Enter a digit: c"
+ echo -e 'Enter a digit: c'
Enter a digit: read var1
+ read var1
4
if ["$var1" -ge 1 -a "$var1" -le 9 ]
   then
      echo "Good input!"
fi
+ '[4' -ge 1 -a 4 -le 9 ]
debug_demo: [4: not found
exit 0
+ exit 0
$ ./debug_demo
Enter a digit: 4
Good input!
$
```

The following in-chapter exercise is designed to enhance your understanding of interrupt processing and the debugging features of the Bourne shell.

EXERCISE 13.8

Test the scripts in the **trap_demo** and **canleave** files on your UNIX system. Do they work as expected? Make sure you understand them. If your versions do not work properly, use the sh -xv command to debug them.

SUMMARY

The Bourne shell does not have the built-in capability for numeric integer data processing in terms of arithmetic, logic, and shift operations. In order to perform arithmetic and logic operations on integer data, the expr command must be used.

The here document feature of the Bourne shell allows standard input of a command in a script to be attached to the data within the script. The use of this feature results in more efficient programs because no extra file-related operations, such as file open and read, are needed, as the data is within the script file and should have been loaded into the main memory when the script was loaded.

The Bourne shell also allows the user to write programs that ignore signals such as keyboard interrupt (<Ctrl+C>). This useful feature can be used, among other things, to disable program termination when it is in the middle of updating a file. The trap command can be used to invoke this feature.

The Bourne shell has powerful I/O features that allow explicit processing of files. The `exec` command can be used to open a file for reading or writing and to associate a small integer, called a file descriptor, with it. The command `exec n< file` opens a file for reading and assigns it a file descriptor `n`. The command line `exec n> file` opens a file for writing and assigns it a file descriptor `n`. This feature allows writing scripts for processing files. The command line `exec n<&-` can be used to close a file with descriptor `n`. The `exec` command provides various other file-related features, including opening a here document and assigning it a file descriptor, which allows the use of a here document anywhere in the script.

The Bourne shell programs can be debugged by using the `-x` and `-v` options of the `sh` command. This technique allows viewing the commands in the user's script after variable substitution but before execution.

QUESTIONS AND PROBLEMS

1. Why is the `expr` command needed?
2. What is the here document? Why is it useful?
3. Write a Bourne shell script `cv` that takes the side of a cube as a command line argument and displays the volume of the cube.
4. Modify the `countup` script in Section 13.2 so that it takes two integer command line arguments. The script displays the numbers between the two integers (including the two numbers) in ascending order if the first number is smaller than the second, and in descending order if the first number is greater than the second. Name the script `count_up_down`.
5. Write a Bourne shell script that prompts you for a user ID and displays your login name, your name, and your home directory.
6. Write a Bourne shell script that takes a list of integers as the command line argument and displays a list of their squares and the sum of the numbers in the list of squares.
7. Write a Bourne shell script that takes a machine name as an argument and displays a message informing you whether the host is on the local network.
8. What are signals in UNIX? What types of signals can be intercepted in the Bourne shell scripts?
9. Write a Bourne shell script that takes a file name and a directory name as command line arguments and removes the file if it is found under the given directory and is a simple file. If the file (the first argument) is a directory, it is removed (including all the files and subdirectories under it).
10. Write a Bourne shell script that takes a directory as an argument and removes all the ordinary files under it that have **.o**, **.ps**, and **.jpg** extensions. If no argument is specified, the current directory is used.

11. Enhance the `diff2` script in Section 13.5 so that it displays the line numbers where the two files differ.
12. Enhance the `diff2` script of Problem 11 so that, if only one file is passed to it as a parameter, it uses standard input as the second file.
13. Write a Bourne shell script for Problem 16 in Chapter 12, but use functions to implement the service code for various options.
14. Suppose that a function, `f`, is defined first, followed by defining `f` as a variable. Would `f` refer to the function or the variable after you have provided the second definition? Why? Show a shell session to support your answer.
15. Write a Bourne shell script that implements the following menu options:
    a. Display the CPU used by your system
    b. Display the name of the operating system used by your computer
    c. Display `a` and `b` on the screen separated by a vertical tab
    d. Display the full pathnames for the commands that have been executed on your system
    e. Display the maximum number of files a process may open
    f. Display the maximum number of simultaneous processes a user may have on the system

Your program should not terminate on signals 1, 2, 3, 15, and 18. Make use of a function to display the program menu.

Hint: Review the man pages for the following commands: `hash`, `ulimit`, `uname`.

CHAPTER 14

Introductory C Shell Programming

Objectives

- To introduce the concept of shell programming
- To describe how C shell programs can be executed
- To explain the concept and use of shell variables in C shell
- To discuss how command line arguments are passed to C shell programs
- To discuss the concept of command substitution
- To describe some basic coding principles
- To write and discuss a few sample C shell scripts
- To cover the commands and primitives

  ```
  *, =, ", ', ', &, <, >, ;, |, \, /, [], (), continue, csh,
  exit, env, foreach, goto, head, if, ls, set, setenv, shift,
  switch, while, unset, unseten
  ```

14.1 INTRODUCTION

The C shell is more than a command interpreter, it has a programming language of its own that can be used to write shell programs for performing various tasks that cannot be performed by any existing command. Shell programs, commonly known as *shell scripts*, in the C shell consist of shell commands to be executed by a shell and are stored in ordinary UNIX files. The shell allows the use of read/write storage places, called *shell variables*, to make it easier for the user or programmer to work and for programmers to use as scratch pads for completing tasks. The C shell also has program control flow commands/

statements that allow the writers of shell scripts to implement multiway branching and repeated execution of a block of commands.

14.2 RUNNING A C SHELL SCRIPT

There are three ways to run a C shell script. The first step for all three methods is to make the script file executable by adding the execute permission to the existing access permissions for the file. You can do so by running the following command, where **script_file** is the name of the file containing the shell script.

```
% chmod u+x script_file
%
```

Clearly, in this case you make the script executable for yourself only. However, you can set appropriate access permissions for the file if you also want other users to be able to execute it. Once you have made the script file executable, you can type `./script _ file` as a command to execute the shell script, as follows:

```
% ./script_file
... Output of the script if any ...
%
```

If your search path (the `path` variable) includes your current directory (.), you can simply use the `script _ file` command, instead of using the `./script _ file` command. For the rest of this chapter, we assume that your `path` variable includes your current directory.

As described in Chapter 10, a child of the current shell process executes the script. Thus, with this method, the script executes properly if you are using the C shell but not if you are using any other shell. If you are currently using some other shell, first execute the `/bin/csh` command to run the C shell and then run the `script _ file` command, as shown in the following example. Here, we assume that your current shell is the Bourne shell (with the `$` prompt). After the script has completed its execution, we press `<Ctrl+D>` to terminate the Bourne shell and return to the C shell.

```
$ /bin/csh
% ./script_file
... Output of the script if any ...
% <Ctrl+D>
$
```

The second method of executing a shell script is to run the `/bin/csh` command with the script file as its parameter. Thus, the following command executes the shell script in **script_file**.

```
% /bin/csh script_file
... Output of the script if any ...
%
```

If your `path` variable includes the **/bin** directory, you can simply use the `csh` command, instead of using the `/bin/csh` command.

The third method, which is also the most commonly used method, is to force the current shell to execute a script in the C shell, regardless of your current shell. You can do so by beginning a shell script with the following line:

```
#!/bin/csh
```

When your current shell encounters the string `#!`, it takes the rest of the line as the absolute pathname for the shell to be executed, under which the script in the file is executed.

Throughout this chapter, we would use the `chmod u+x script _ file` command to make **script_file** executable by the owner of the file and run the script by using the `./script _ file` command.

14.3 SHELL VARIABLES AND RELATED COMMANDS

A *variable* is a main memory location that is given a name. This allows you to reference the memory location by using its name instead of its address. The name of a shell variable is comprised of digits, letters, and underscores, with the first character being a letter or underscore. Because the main memory is read/write storage, you can read a variable's value or assign it a new value. Under the C shell, the value of a variable can be a string of characters or a numeric value. There is no theoretical limit to the length of a variable's value stored as a string; the practical limit is dictated by the length of a line.

Shell variables can be one of two types: *shell environment variables* and *user-defined variables*. You can use environment variables to customize the environment in which your shell runs and for proper execution of shell commands. A copy of these variables is passed to every command that executes in the shell as its child. Most of these variables are initialized when the login script(s) execute(s), according to the environment set by your system administrator. You can further customize your environment by assigning appropriate values to some or all of these variables in your **~/.login** and **~/.cshrc** start-up files, which also execute when you log on. See Chapter 2 for a discussion of start-up files. Table 14.1 lists most of the environment variables whose values you can change.

The shell environment variables listed in Table 14.1 are *writable*, meaning that you can assign them any values to make your shell environment meet your needs. Other shell environment variables are *read only*. That is, you can use (read) the values of these variables, but you cannot change them directly. These variables are most useful for processing command line arguments (also known as *positional arguments*), the parameters passed to a shell script at the command line. Examples of command line arguments are the source and destination files in the `cp` command. Some other read-only shell variables are used to keep track of the process ID of the current process, the process ID of the most recent background process, and the exit status of the last command. Some important *read-only shell environment variables* are listed in Table 14.2.

User-defined variables are used within shell scripts as temporary storage places whose values can be changed when the program executes. These variables can be made global and

TABLE 14.1 Some Important Writable C Shell Environment Variables

Environment Variable	Purpose of the Variable
`cdpath`	Directory names that are searched, one by one, by the cd command to find the directory passed to it as a parameter; the cd command searches the current directory if this variable is not set
`home`	Name of your home directory, the C shell places you in this directory when you first log on
`mail`	Name of your system mailbox file
`path`	Variable that contains your search path—the directories that a shell searches to find an external command or program that you try to run
`prompt`	Primary shell prompt that appears on the command line, usually set to `%`
`prompt2`	Secondary shell prompt displayed on the second line of a command if the shell thinks that the typed command is not complete, typically when the command line terminates with a backslash (\), the escape character
`cwd`	Name of the current working directory
`term`	Type of user's console terminal

TABLE 14.2 Some Important Read-Only C Shell Environment Variables

Environment Variable	Purpose of the Variable
`$argv[0]` or `$0`	Name of the executing program
`$number or ${number}`	Equivalent to `$argv[number]`, where number may be 0, 1, 2, …
`$argv[*] or $*`	Values of all of the command line arguments
`$#argv or $#`	Total number of command line arguments
`$$`	Process ID (PID) of the current process; typically used as a file name extension to create (most probably) unique file names
`$!`	PID of the most recent background process

passed to the commands that execute in the shell script in which they are defined. As with most programming languages, you have to declare C shell variables before using them. A reference to a C shell variable that has not been declared results in an error.

You can display the values of all shell variables (environmental and user-defined) and their current values by using the `set` command without any argument. The following is a sample run of the `set` command on a machine running PC-BSD.

```
% set
_
addsuffix
anyerror
argv ()
autocorrect
autoexpand
autolist    ambiguous
autorehash
complete    enhance
correct     cmd
csubstnonl
```

```
cwd  /home/sarwar
dirstack    /home/sarwar
echo_style  bsd
edit
euid 1004
euser       sarwar
filec
gid  1008
group       faculty
history     100
home /home/sarwar
killring    30
loginsh
mail /var/mail/sarwar
owd
path (/usr/local/share/pcbsd/bin /sbin /bin /usr/sbin /usr/bin
/usr/games /usr/pbi/bin /usr/local/sbin /usr/local/bin /home
/sarwar/bin .)
prompt      %{\033]0;%n@%m:%~\007%}[%B%n@%m%b] %B%~%b%#
prompt2     %R?
prompt3     CORRECT>%R (y|n|e|a)?
savehist    100
shell       /bin/csh
shlvl       1
status      0
tcsh6.18.01
term xterm-color
tty  pts/1
uid  1004
user sarwar
version     tcsh 6.18.01 (Astron) 2012-02-14 (x86_64-amd-FreeBSD)
options wide,nls,dl,al,kan,sm,rh,color,filec
%
```

The @ command displays the same information. You can use the env (System V) and printenv (BSD) commands to display both the environment variables and their values. The following is a sample output of the env command on the same machine that we ran the set command on.

```
% env
USER=sarwar
LOGNAME=sarwar
HOME=/home/sarwar
MAIL=/var/mail/sarwar
PATH=/usr/local/share/pcbsd/bin:/sbin:/bin:/usr/sbin:/usr/bin:/
usr/games:/usr/pbi/bin:/usr/local/sbin:/usr/local/bin:/home/
sarwar/bin:.
```

```
TERM=xterm-color
BLOCKSIZE=K
SHELL=/bin/csh
SSH_CLIENT=182.185.191.114 52885 22
SSH_CONNECTION=182.185.191.114 52885 202.147.169.196 22
SSH_TTY=/dev/pts/1
HOSTTYPE=FreeBSD
VENDOR=amd
OSTYPE=FreeBSD
MACHTYPE=x86_64
SHLVL=1
PWD=/home/sarwar
GROUP=faculty
HOST=pcbsd-srv
REMOTEHOST=182.185.191.114
LANG=en_US.UTF-8
EDITOR=vi
PAGER=more
MANPATH=/usr/share/man:/usr/local/man:/usr/share/openssl/man:/usr/
pbi/man:/usr/local/lib/perl5/5.16/man:/usr/local/lib/perl5/5.16/
perl/man
NO_PROXY=127.0.0.1,localhost
no_proxy=127.0.0.1,localhost
CLICOLOR=true
MORE=-erX
%
```

In the following in-chapter exercises, you will create a simple shell script and make it executable. Also, you will use the `set` and `env` commands to display the names and values of shell variables in your environment.

EXERCISE 14.1

Display the names and values of all shell variables on your UNIX machine. What command(s) did you use?

EXERCISE 14.2

Create a file that contains a shell script comprising the `date` and `who` commands, one on each line. Make the file executable and run the shell script. List all the steps for completing this task.

14.4 READING AND WRITING SHELL VARIABLES

You can use any of three commands to assign a value to (write) one or more shell variables (environmental or user-defined): `@`, `set`, and `setenv`. The `set` and `setenv` commands are used to assign a string to a variable. The difference is that the `setenv` command declares and initializes a global variable, whereas the `set` command declares and initializes a local variable. You can use the `@` command to assign an integer value to a local variable. The following are brief descriptions of the `@` and `set` commands. We describe the `setenv` command in Section 14.3.3.

SYNTAX

```
set [variable1[=strval1] variable2 [=strval2] … variableN [=strvalN]]
@ [variable1[=numval1] variable2 [=numval2] … variableN [=numvalN]]
```

Purpose: Assign values **`strval1`**, …, **`strvalN`** or **`numval1`**, …, **`numvalN`** to variables **`variable1`**, …, **`variableN`**, respectively, where a value can be **`strval`** for a string value and **`numval`** for a numeric value

No space is required before or after the = sign for the `@` and `set` commands, but spaces can be used for clarity. If a value contains spaces, you must enclose the value in parentheses. The `set` command with only the name of a variable declares the variable and assigns it a null value. Unlike the Bourne shell, where every variable is automatically initialized, in the C shell you must declare a variable in order to initialize and use it. Without any arguments, the `set` and `@` commands display all shell variables and their values. Multiword values are displayed in parentheses. (We discuss the `@` command in detail in Chapter 15.) You can refer to (i.e., read) the current value of a variable by inserting a dollar sign ($) before the name of a variable. You can use the `echo` command to display values of shell variables.

In the following session, we show how shell variables can be read and written.

```
% echo $name
name: Undefined variable.
% set name
% echo $name

% set name = John
% echo $name
John
% set name = John Doe II
% echo $name
John
% echo $Doe $II

% set name = (John Doe)
% echo $name
John Doe
% set name = John*
% echo $name
John.Bates.letter John.Johnsen.memo John.email
% set name = "John*"
% echo "$name"
John*
```

```
% echo "The name $name sounds familiar!"
The name John* sounds familiar!
% echo \$name
$name
% echo '$name'
$name
```

The preceding session shows that, if values that include spaces are not enclosed in quotes, the shell assigns the first word to the variable and the remaining as null-initialized variables. In other words, the command `set name = John Doe II` initializes the `name` variable to John, and declares `Doe` and `II` as string variables initialized to null. When you display a variable initialized to null, the shell displays a blank line. Also, after the `set name=John*` command has been executed and `$name` is not enclosed in quotes in the `echo` command, the shell lists the file names in your present working directory that match `John*`, with `*` considered as the shell metacharacter—that is, files that start with the string `John`, followed by 0 or more characters. If your current directory does not contain any files that start with the string `John`, the `set name = John*` command returns an error message, `set: No match.`, and a subsequent `echo $name` command would display the previous value of the name variable, `John Doe` in this case. Running the `echo *` command would display the names of all the files in your current directory. The preceding session also shows that single quotes can be used to process the value of the `name` variable literally. In fact, you can use single quotes to process the whole string literally. The backslash character can be used to escape the special meaning of any single character, including `$`, and treat it literally. In the C shell, in order to escape the special meaning of `!`, you must use a backslash (\) `before the !` symbol if it is followed by any character other than a space.

A command consisting of `$variable` alone results in the value of `variable` being executed as a shell command. If the value of `variable` comprises a valid command, the expected results are produced. If `variable` does not contain a valid command, the shell, as expected, displays an appropriate error message. The following session illustrates this point with examples. The variable used in this session is `command`.

```
% set command = pwd
% $command
/usr/home/sarwar/unix3e/ch14
% set command = hello
% $command
hello: Command not found.
%
```

14.4.1 Command Substitution

When a command is enclosed in back quotes (also known as *grave accents*), the shell executes the command and substitutes the output of the command for the command (including back quotes). This process is referred to as command substitution. The following is a brief description of command substitution.

SYNTAX

'command'

Purpose: Execute **command** and substitute its output for **'command'**

The next session illustrates the concept. In the first assignment statement, the variable called `command` is assigned the value `pwd`. In the second assignment statement, the output of the `pwd` command is assigned to the `command` variable.

```
% set command = pwd
% echo "The value of command is: $command."
The value of command is: pwd.
% set command = 'pwd'
% echo "The value of command is: $command."
The value of command is: /usr/home/sarwar/unix3e/ch14.
%
```

Command substitution can be specified in any command. For example, in the following command line, the output of the `date` command is substituted for `'date'` before the `echo` command is executed.

```
% echo "The date and time are 'date'."
The date and time are Sun Sep 7 14:12:13 PKT 2014.
%
```

The following in-chapter exercises are designed to reinforce the creation and use of shell variables and the concept of command substitution.

EXERCISE 14.3

Assign your full name to a shell variable `myname` and echo its value. How did you accomplish the task? Show your work.

EXERCISE 14.4

Assign the output of the command `echo "Hello, world!"` to the `myname` variable and then display the value of `myname`. List the commands that you executed to complete this task.

14.4.2 Exporting Environment

When a variable is created, it is not automatically known to subsequent shells. The `setenv` command passes the *value* of a variable to subsequent shells. Thus, when a shell script is called and executed from another shell script, it does not get automatic access to the variables defined in the original (caller) script unless they are explicitly made available to it. You can use the setenv command to assign a value to a string variable and pass the value of the variable to subsequent commands that execute as children of the script. Because all read/write shell environment variables are available to every command, script, and subshell, they are initialized by the setenv command. The setenv command is equivalent to creating a variable, followed by exporting it with the export command to make its value available to all subsequent shells in the Bourne shell. The following is a brief description of the setenv command.

SYNTAX

setenv [variable [strval]]

Purpose: Assigns to variable a string value **strval** and exports **variable** and a copy of its value so that it is available to every command executed from this point on

The following session shows a simple use of the setenv command. The name variable is initialized to John Doe and is exported to subsequent commands executed under the current shell and any subshell that runs under the current shell. Note that unlike the set command, the setenv command requires that you enclose multiword values in double quotes, not in parentheses.

```
% setenv name "John Doe"
% echo $name
John Doe
%
```

The next session illustrates the concept of exporting shell variables via some simple shell scripts.

```
% cat display_name
#!/bin/csh
echo $name
exit 0
%
% set name=(John Doe)
% ./display_name
name: Undefined variable.
%
```

Note that the script in the **display_name** file displays an undefined variable error message, even though we initialized the name variable just before executing this script. The reason is that the name variable declared interactively is not exported before running the script, and the name variable used in the script is local to the script. As this local variable name is uninitialized, the echo command displays the error message. As stated before, unlike the Bourne shell, the C shell requires declaration of a variable before its use.

You can use the exit command to transfer control out of the executing program and pass it to the calling process, the current shell process in the preceding session. The only argument of the exit command is an optional integer number that is returned to the calling process as the exit status of the terminating process. All UNIX commands return an exit status of zero upon *success*—that is, after successfully performing their tasks, and nonzero upon *failure*. The return status value of a command is stored in the read-only environment variable $? and can be checked by the calling process. In shell scripts, depending on the task at hand, the status of a command execution can be checked and then subsequent action can be taken. Later in the chapter we show the use of the read-only environment variable $? in some shell scripts. When the exit command is executed without an argument, the UNIX kernel sets the return status value for the script.

In the following session, the name variable is exported after it has been initialized, thus making it available to the display _ name script. The session also shows that the return status of the display _ name script is 0, implying successful execution of display _ name.

```
% setenv name "John Doe"
% ./display_name
John Doe
% echo $?
0
%
```

We now show that a copy of an exported variable's value is passed to any subsequent command. That is, a command has access only to the value of an exported variable; it cannot assign a new value to the variable. Consider the following script in the **export_demo** file.

```
% cat export_demo
#!/bin/csh
setenv name "John Doe"
display_change_name
echo "$name"
% cat display_change_name
#!/bin/csh
echo "$name"
set name = (Plain Jane)
echo "$name"
exit 0
% ./export_demo
John Doe
Plain Jane
John Doe
%
```

When the `export_demo` script is invoked, the `name` variable is set to `John Doe` and exported so that it becomes part of the environment of the commands that execute under `export_demo` as its children. The first `echo` command in the `display_change_name` script displays the value of the exported variable `name`. It then initializes a local variable called `name` to `Plain Jane`. The second `echo` command therefore echoes the current value of the local variable `name` and displays `Plain Jane`. When the `display_change_name` script finishes, the `display_name` script executes the `echo` command and displays the value of the exported `name` variable, thus displaying `John Doe`.

14.4.3 Resetting Variables

A variable retains its value for as long as the script in which it is initialized executes. You can remove a variable from the environment by using the `unset` and `unsetenv` commands. The following are brief descriptions of the commands.

SYNTAX

```
unset variable-list
unsetenv variable
```

Purpose: Remove the specified variables from the environment. The variables in **variable-list** are separated by spaces. The **`unset`** command is used for the variables declared by the `set` or `@` commands. The **`unsetenv`** command is used for variables declared by the `setenv` command.

Next we show a simple use of the `unset` command. The variables `name` and `place` are set to `John` and `Corvallis`, respectively, and the `echo` command displays the values of these variables. The `unset` command resets the `name` variable, defined with the `set` command, to null. Thus, the `echo "$name"` command should display a message saying that the `name` variable is undefined. However, the command shows `John Doe` as its output. If you go back and look at the shell session before the last one, you would see that `name` was initialized to `John Doe` using the `setenv name "John Doe"` command. The `unset name` command in the following session undefined the `name` variable defined by using this command, but the `name` variable defined using the `setenv` command remained defined with the value `John Doe`. When we run the `unsetenv name` command in the following session, the `name` variable defined with the `setenv name "John Doe"` command is undefined and the `echo "$name"` command shows the error message `"name: Undefined variable."`.

```
% set name=John place=Corvallis
% echo "$name $place"
John Corvallis
% unset name
% echo "$name"
John Doe
```

```
% echo "$place"
Corvallis
% unsetenv name
% echo "$name"
name: Undefined variable.
$
```

The following command removes the variables `name` and `place`, defined with the `set` command, from the environment:

```
% unset name place
%
```

To reset a variable, explicitly assign it a null value by using the `set` command with the variable name only. Or you can assign the variable no value and simply hit `<Enter>` after the = sign, as in.

```
% set country=
% echo "$country"

% set place
% echo $place
%
```

Here, the `set` command is used to reset the `country` and `place` variables to null.

14.4.4 Reading from Standard Input

So far, we have shown how you can assign values to shell variables statically at the command line or by using the assignment statement. If you want to write an interactive shell script that prompts the user for keyboard input, you need to use the `set` command in order to store the user input in a shell variable, according to the following syntax.

SYNTAX

```
set variable = $<
set variable = 'head -1'
```

Purpose: Read one line from **`stdin`** into **`variable`**; note the use of back quotes (grave accents) in **`'head -1'`**

These commands allow you to read one line of keyboard input into variable. Some C shell implementations do not support the first `set` command, but the second `set` command works in all implementations. You can use the syntax for the second command to assign the first line of keyboard input to variable. Unlike that of the Bourne shell, the keyboard input feature of the C shell does not allow assignment of words in a line to multiple variables. However, the words in a line are stored in the form of an array, and you can access them by using the name of the variable (we discuss arrays in Chapter 15).

We illustrate the semantics of the second set command with a script in the **keyin_demo** file, as follows:

```
% cat keyin_demo
#!/bin/csh
echo -n "Enter input: "
set line = 'head -1'
echo "You entered: $line"
exit 0
%
```

In the following run, enter `UNIX rules the network computing world!`. The set command takes the whole input and puts it in the shell variable line without the newline character. The output of the echo command displays the contents of the shell variable line.

```
% ./keyin_demo
Enter input: UNIX rules the network computing world!
You entered: UNIX rules the network computing world!
%
```

The -n option is used with the echo command to keep the cursor on the same line. If you do not use this option, the cursor moves to the next line, which is what you want to see happen while displaying information and error messages. However, when you prompt the user for keyboard input, you should keep the cursor in front of the prompt for a more user-friendly interface. The C shell on most UNIX systems that are based on BSD UNIX, runs the BSD version of the echo command, which does not support the \c option and other options supported by the System V version of the echo command. The BSD version does support Standard ASCII control sequences that can be used to display other special characters, such as <Ctrl+H> for backspace and <Ctrl+G> for bell.

In the following in-chapter exercise, you will use the set command to practice reading from **stdin** in shell scripts.

EXERCISE 14.5

Write commands for reading a value into the myname variable from the keyboard and exporting it so that the commands executed in any child shell have access to the variable.

14.5 PASSING ARGUMENTS TO SHELL SCRIPTS

In this section, we describe how command line arguments can be passed to shell scripts and manipulated by them. As we discussed in Section 14.3, you can pass command line arguments, also called *positional parameters*, to a shell script. The values of these arguments can be referenced by using the names $argv[number] where number may assume values 0, 1, 2, and so on, with $argv[0] referring to the program (or command) name. The positional parameters may also be specified by using the $number or ${number} notation, such as $0, $1, $2, and so on, or ${0}, ${1}, ${2}, and so on. The curly braces are used to isolate number from the character(s) that may follow. If a positional argument referenced in your script is not passed as an argument, it is initialized to a value of null. However, while displaying such arguments, the $number notation works fine, but the $argv[number] notation results in an error message, argv: Subscript out of range., which simply means that you are indexing the argv array for an element that does not exist. You can use the names $#argv or $# to refer to the total number of arguments passed in an execution of the script. The names $argv[*], argv, or $* refer to the values of all of the arguments. The names $argv[0] and $0 refer to the name of the script file (i.e., the command name). In the following session, we use the shell script in the **cmdargs_demo** file to show how you can use these variables.

```
% cat cmdargs_demo
#!/bin/csh
echo "The command name is $0."
echo "The number of command line arguments is $#argv."
echo -n "The values of the command line arguments are: "
echo "$1 $2 $3 $4 $5 $6 $7 $8 $9 $10 $11 $12"
echo "Another way to display command line arguments is: $argv[*]"
echo "Yet another is: $*"
exit 0
% ./cmdargs_demo 1 2 3 4 5 6 7 8 9 10 11 12
The command name is cmdargs_demo.
The number of command line arguments is 12.
The values of the command line arguments are:1 2 3 4 5 6 7 8 9 10 11 12
Another way to display command line arguments is:1 2 3 4 5 6 7 8 9 10 11 12
Yet another is: 1 2 3 4 5 6 7 8 9 10 11 12
% ./cmdargs_demo One Two 3 Four 5 6
The command name is cmdargs_demo.
The number of command line arguments is 6.
The values of the command line arguments are: One Two 3 Four 5 6
Another way to display command line arguments is: One Two 3 Four 5 6
Yet another is: One Two 3 Four 5 6
%
```

The C shell maintains as many as command line arguments at a time as the command line length allows. You can write scripts that handle command line arguments, one at a time, by shifting the arguments left by one argument. To do so, use the shift command. By default, this command shifts the command line arguments to the left by one position, moving $argv[2] to $argv[1], $argv[3] become $argv[2], and so on. The first argument, `$argv[1]`, is shifted out. Once shifted, the arguments cannot be restored to their original values. More than one position can be shifted if specified as an argument to the command. The following is a brief description of the command.

SYNTAX

`shift [variable]`

Purpose: Shift the words in **`variable`** one position to the left; if no variable name is specified, the command line arguments are assumed

The script in the **shift_demo** file shows the semantics of the `shift` command with the implicit variable, the command line arguments. The `shift` command shifts the first argument out and the remaining arguments to the left by one position. The three `echo` commands are used to display the current values of program names, all positional arguments (`$#argv[*]`), and the values of the first three positional parameters, respectively. The results of execution of the script are obvious.

```
% cat shift_demo
#!/bin/csh
echo "The name of the program is $0."
echo "The arguments are: $argv[*]."
echo "The first three arguments are: $argv[1] $argv[2] $argv[3]."
shift
echo "The name of the program is $0."
echo "The arguments are: $argv[*]."
echo "The first three arguments are: $argv[1] $argv[2] $argv[3]"
exit 0
% ./shift_demo 1 2 3 4 5 6 7 8 9 10 11 12
The name of the program is shift_demo.
The arguments are: 1 2 3 4 5 6 7 8 9 10 11 12.
The first three arguments are: 1 2 3.
The name of the program is shift_demo.
The arguments are: 2 3 4 5 6 7 8 9 10 11 12.
The first three arguments are: 2 3 4
%
```

The values of positional arguments can be altered by using the `set` command with `argv` as its argument. The most effective use of this command is in conjunction with command substitution. The following is a brief description of the command.

SYNTAX

`set argv = [argument-list]`

Purpose: Set values of the positional arguments to the arguments in **argument-list**

The following is a simple interactive use of the command. The `date` command is executed to show that the output has six fields. The `set argv = 'date'` command sets the positional parameters according to the output of the `date` command. In particular, `$argv[1]` is set to `Sat`, `$argv[2]` to `Aug`, `$argv[3]` to `7`, `$argv[4]` to `13:26:42`, `$argv[5]` to `PDT`, and `$argv[6]` to `2004`. The `echo $argv[*]` command displays the values of all positional arguments. The third `echo` command displays the date in a commonly used form.

```
% date
Sun Sep  7 08:25:04 PKT 2014
% set argv = 'date'
% echo $argv[*]
Sun Sep 7 08:25:17 PKT 2014
% echo "$argv[2] $argv[3], $argv[6]"
Sep 7, 2014
%
```

The script in **set_demo** shows another use of the command. When the script is run with a file argument, it generates a line that contains the file name, the file's inode number, and the file size (in bytes). The `set` command is used to assign the output of `ls -il` command as the new values of the positional arguments `$argv[1]` through `$argv[9]`. We show the output of the `ls -il` command in case you do not remember the format of the output of this command.

```
% cat set_demo
#!/bin/csh
set filename = $argv[1]
set argv = 'ls -il $filename'
set inode = $argv[1]
set perms = $argv[2]
set size = $argv[6]
echo "File Name:        $filename"
echo "Inode Number:     $inode"
echo "Permissions:      $perms"
echo "Size (bytes):     $size"
exit 0
% ./set_demo lab3
File Name:    lab3
Inode Number: 4313
Permissions:  -rw-r--r--
Size (bytes): 221
% ls -il lab3
4313 -rw-r--r-- 1 sarwar sarwar 221 Sep 7 08:30 lab3
%
```

In the following in-chapter exercises, you will use the `set` and `shift` commands in order to reinforce the use and processing of command line arguments.

EXERCISE 14.6

Write a shell script that displays all command line arguments, shifts them to the left by two positions, and redisplays them. Show the script along with a few sample runs.

EXERCISE 14.7

Update the shell script in Exercise 14.1 so that, after accomplishing this task, it sets the positional arguments to the output of the `who | head -1` command and displays the positional arguments again.

14.6 COMMENTS AND PROGRAM HEADERS

You should develop the habit of putting comments in your programs to describe the purpose of a particular series of commands. At times, you should even briefly describe the purpose of a variable or assignment statement. Also, you should use a program header for every shell script that you write. These are simply good software engineering practices. A *program header* is a set of introductory comments used to explain the script. Program header and in-code comments help a programmer who has been assigned the task of maintaining (i.e., modifying or enhancing) your code to understand it quickly. They also help you understand your own code, in particular, if you reread it after some period of time. Long ago, putting comments in the program code or creating separate documentation for programs was not a common practice. Such programs when inherited by a programmer or a team are very difficult to understand and maintain, and are commonly known as *legacy code*. You may find different definitions for legacy code in the literature.

A good program header must contain at least the following items. In addition, you can insert any other items that you feel are important or are commonly used in your organization or group as part of its coding rules.

1. Name of the file containing the script
2. Name of the author
3. Date written
4. Date last modified
5. Purpose of the script (in one or two lines)
6. A brief description of the algorithm used to implement the solution to the problem at hand

A comment line, including every line in the program header, must start with the number sign (#), as in.

```
# This is a comment line.
```

However, a comment does not have to start at a new line; it can follow a command, as in

```
set Var1=a Var2 Var3=b  # Assign "a" to Var1, "b" to Var3, and
                        # declare a variable Var2 with an initial
                        # value of  null.
```

The following is a sample header for the **set_demo** script.

```
# File Name:            ~/Cshell/examples/set_demo
# Author:               Syed Mansoor Sarwar
# Date Written:         August 10, 1999 (by the original author)
# Modified:             May 21, 2004 (by the original author)
# Last Modified         September 7, 2014
# Purpose:              To illustrate how the set command works
# Brief Description:    The script runs with a filename as the only
#                       command line argument, saves the filename,
#                       runs the set command to assign the output
#                       of the ls -il command to positional
#                       arguments ($1–$9), and displays file name,
#                       its inode number, file permissions, and
#                       file size (in bytes).
```

We do not show the program headers for all of the sample scripts in this textbook for the sake of brevity.

14.7 PROGRAM CONTROL FLOW COMMANDS

The program control flow commands/statements are used to determine the sequence in which statements in a shell script execute. The four basic types of statements for controlling the flow of a script are two-way branching, multiway branching, repetitive execution of a group of commands/statements, and transferring control to a particular statement via a *jump* or *goto* statement of some sort. The C shell statement for two-way branching is the `if` statement, the statements for multiway branching are the `if` and `switch` statements, and the statements for repetitive execution of some code are the `foreach` and `while` statements. In addition, the C shell has a `goto` statement that allows you to jump to any command in a program.

14.7.1 The if-then-else-endif Statement

The most basic form of the `if` statement is used for one-way branching, but the statement can also be used for multiway branching. The following is a brief description of the statement. The words in monospace type are *keywords* and must be used as shown in the syntax. Everything in brackets is optional. All the command lists are designed to help you accomplish the task at hand.

SYNTAX

```
if (expression) then
    then-command-list
   [else if (expression) then
     then-command-list
   …
   else
     else-command-list]
endif
```

Purpose: To implement two-way or multiway branching

Here, an **expression** is a list of commands. The execution of commands in **expression** returns a status of true (success) or false (failure). We discuss three versions of the **if** statement that together comprise the statement's complete syntax and semantics. The most basic use of the **if** statement is without any optional features and results in the following syntax for the statement, which is commonly used for two-way branching.

SYNTAX

```
if (expression) then
   then-commands
endif
```

Purpose: To implement two-way branching

If **expression** is true, the **then-commands** are executed; otherwise, the command after, **endif** is executed. The semantics of the statement are shown in Figure 14.1.

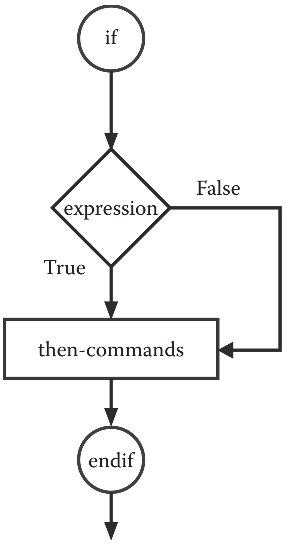


FIGURE 14.1 Semantics of the `if-then-if` statement.

You can form an expression by using a variety of operators for testing files, testing and comparing integers and strings, and logically connecting two or more expressions to form complex expressions. Table 14.3 describes the operators that can be used to form expressions, along with their meanings. Operators not related to files are listed in the order of their precedence (from high to low): parentheses, unary, arithmetic, shift, relational, bitwise, and logical.

We use the preceding syntax of the `if` command to modify the script in the **set_demo** file so that it takes one command line argument only and checks on whether the argument is a file or a directory. The script returns an error message if the script is run with none or more than one command line argument, or if the command line argument is not an

TABLE 14.3 C Shell Operators for Forming Expressions

Operator	Function	Operator	Function
Parentheses		**Relational Operators**	
`()`	To change the order of evaluation	`>`	Greater than
		`<`	Less than
		`>=`	Greater than or equal to
		`<=`	Less than or equal to
		`!=`	Not equal to (for string comparison)
		`==`	Equal to (for string comparison)
Unary Operators		**Bitwise Operators**	
`-`	Unary minus	`&`	AND
`~`	One's complement	`^`	XOR (exclusive OR)
`!`	Logical negation (NOT)	`|`	OR
Arithmetic Operators		**Logical Operators**	
`%`	Remainder	`&&`	AND
`/`	Divide	`||`	OR
`*`	Multiply		
`-`	Subtract		
`+`	Add		
Shift Operators			
`>>`	Shift right		
`<<`	Shift left		

File- and String-Related Operators

Operator	*Function*	*Operator*	*Function*	*Operator*	*Function*
`-d` **file**	True if **file** is a directory	`-e` **file**	True if **file** exists	`-f` **file**	True if **file** is ordinary **file**
`-o` **file**	True if user owns **file**	`-r` **file**	True if **file** is readable	`-w` **file**	True if **file** is writable
`-x` **file**	True if **file** is executable	`-z` **file**	True if length of **file** is zero bytes		

ordinary file. The name of the script file is **if_demo1**. The contents of the file and its sample runs are as follows.

```
% cat if_demo1
#!/bin/csh
if ( ( $#argv == 0 ) || ( $#argv > 1 ) ) then
    echo "Usage: $0 ordinary_file"
    exit 1
endif
if ( ! -e $1 ) then
    echo "$1 : non-existent file"
    exit 1
endif
if ( -d $1 ) then
    echo "$1 : directory"
    echo "Usage: $0 ordinary_file"
    exit 1
endif
if ( -f $1 ) then
    set filename = $argv[1]
    set fileinfo = 'ls -il $filename'
    set inode = $fileinfo[1]
    set perms = $fileinfo[2]
    set size = $fileinfo[6]
    echo "File Name:        $filename"
    echo "Inode Number:     $inode"
    echo "Permissions:      $perms"
    echo "Size (bytes):     $size"
    exit 0
endif
echo "$0: argument must be an ordinary file"
exit 1
% ./if_demo1
Usage: if_demo1 ordinary_file
% ./if_demo1 lab1
lab1 : non-existent file
% ./if_demo1 dir1
dir1 : directory
Usage: if_demo1 ordinary_file
% ./if_demo1 lab3
File Name:         lab3
Inode Number:      4313
Permissions:       -rw-r--r--
Size (bytes):      221
%
```

In the preceding script, the first instance of the `if` statement that contains a compound expression displays an error message and exits the program if you run the script without a command line argument or with more than one argument. The second and third `if` statements, respectively, check if the file specified as command line argument exists and is not a directory. The fourth `if` statement is executed if you run the script with only one command line argument that exists in your directory hierarchy and is not a directory. It produces the desired results if the command line argument is an ordinary file. If the passed argument is not an ordinary file, the condition for the second `if` statement is false and the error message `if _ demo1: argument must be an ordinary file` is displayed. Note that the exit status of the script is `1` when it exits because of an erroneous condition, and `0` when the script executes successfully and produces the desired results.

An important practice in script writing is to correctly indent the commands/statements in it. Proper indentation of programs enhances their readability, making them easier to understand, debug, and maintain (add or remove features). Note the indentation style used in the sample scripts presented in this textbook and follow it when you write scripts.

The second instance of the `if` statement syntax also allows two-way branching. The following is a brief description of the statement.

SYNTAX

```
if (expression) then
   then-commands
else
   else-commands
endif
```

Purpose: To implement two-way branching

If **expression** is true, the commands in **then-commands** are executed; otherwise, the commands in **else-commands** are executed, followed by the execution of the first command after **endif**. The semantics of the statement are shown in Figure 14.2.

We rewrite the **if_demo1** program by using the `if-then-else-endif` statement at the end. The resulting script is in the **if_demo2** file, as shown in the following session. Notice that the program looks cleaner and more readable.

```
% cat if_demo2
#!/bin/csh
if ( ( $#argv == 0 ) || ( $#argv > 1 ) ) then
    echo "Usage: $0 ordinary_file"
    exit 1
endif
if ( ! -e $1 ) then
    echo "$1 : non-existent file"
    exit 1
```

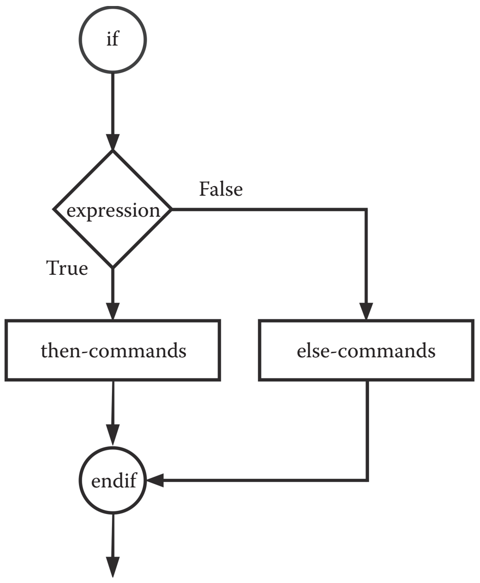


FIGURE 14.2 Semantics of the `if-then-else-endif` statement.

```
endif
if ( -d $1 ) then
    echo "$1 : directory"
    echo "Usage: $0 ordinary_file"
    exit 1
endif
if ( -f $1 ) then
    set filename = $argv[1]
    set fileinfo = 'ls -il $filename'
    set inode = $fileinfo[1]
    set perms = $fileinfo[2]
    set size = $fileinfo[6]
    echo "File Name:        $filename"
    echo "Inode Number:     $inode"
    echo "Permissions:      $perms"
    echo "Size (bytes):     $size"
    exit 0
else
    echo "$0: argument must be an ordinary file"
    exit 1
endif
%
```

The third version of the `if` statement is used to implement multiway branching. The following is a brief description of the statement.

SYNTAX

```
if (expression1) then
   then-commands
else if (expression2) then
   else-if1-commands
else if (expression3) then
   else-if2-commands
...
else
   else-commands
endif
```

Purpose: To implement multiway branching

If **expression1** is true, the commands in **then-commands** are executed. If **expression1** is false, **expression2** is evaluated, and if it is true, the commands in **else-if1-commands** are executed. If **expression2** is false, **expression3** is evaluated. If **expression3** is true, **else-if2-commands** are executed. If **expression3** is also false, the commands in **else-commands** are executed. The execution of any command list is followed by the execution of the command after **endif**. You can use any number of `else-ifs` in an `if` statement to implement multiway branching. The semantics of the statement are illustrated in Figure 14.3.

We enhance the script in the **if_demo2** script so that if the command line argument is a directory, the program displays the number of files and subdirectories in it, excluding hidden files. If the directory is empty, the script informs you so. Implementation also involves the use of the `if-then-else-endif` statement throughout. The resulting script is in the

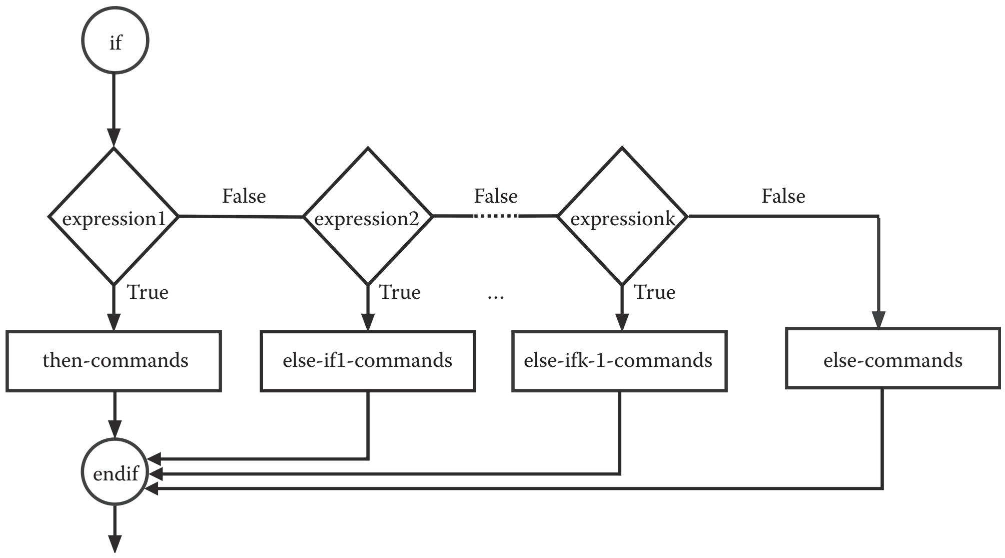


FIGURE 14.3 Semantics of the `if-then-else-if-else-endif` statement.

if_demo3 file, as shown in the following session. As you can see, although it contains more functionality, it is more elegant than the previous two versions.

```
% cat if_demo3
#!/bin/csh
if ( ( $#argv == 0 ) || ( $#argv > 1 ) ) then
    echo "Usage: $0 ordinary_file"
else if ( ! -e $1 ) then
    echo "$1 : non-existent file"
else if ( -d $1 ) then
    set nfiles = 'ls $1 | wc -w'
        if ($nfiles == 0) then
            echo "$1 : Empty directory"
        else
            echo "The number of files in $1 is : $nfiles."
       endif
else if ( -f $1 ) then
    set filename = $argv[1]
    set fileinfo = 'ls -il $filename'
    set inode = $fileinfo[1]
    set perms = $fileinfo[2]
    set size = $fileinfo[6]
    echo "File Name:        $filename"
    echo "Inode Number:     $inode"
    echo "Permissions:      $perms"
    echo "Size (bytes):     $size"
else
    echo "$0: argument must be an ordinary file"
endif
% ./if_demo3
Usage: if_demo3 ordinary_file
% ./if_demo3 lab1
lab1 : non-existent file
% ./if_demo3 dir1
dir1 : Empty directory
% ./if_demo3 .
The number of files in . is : 17.
% ./if_demo3 lab3
File Name:        lab3
Inode Number:     4313
Permissions:      -rw-r--r--
Size (bytes):     221
%
```

If the command line argument is an existing file, the required file-related data is displayed. If the argument is a directory, the number of files in it (including directories and hidden files) is saved in the `nfiles` variables and displayed. If the argument is a nonexistent file or directory, the error message `if _ demo3: argument must be an existing file or directory` is displayed. The sample runs of the script show these cases. The same runs also show the expected outputs when the script is run without an argument and more than one argument.

In the following in-chapter exercises, you will practice the use of `if` statement, command substitution, and manipulation of positional parameters.

EXERCISE 14.8

Create the **if_demo2** script file and run it with no argument, more than one argument, and one argument only. While running the script with one argument, use a directory as the argument. What happens? Does the output of the script make sense?

EXERCISE 14.9

Write a shell script whose single command line argument is a file. If you run the program with an ordinary file, the program displays the owner's name and last update time for the file. If the program is run with more than one argument, it generates meaningful error messages.

14.7.2 The foreach Statement

The `foreach` statement is the first of two statements available in the C shell for repetitive execution of a block of commands in a shell script. These statements are commonly known as *loops*. The following is a brief description of the statement.

SYNTAX

```
foreach variable (argument-list)
   command-list
end
```

Purpose: To execute commands in **command-list** as many times as the number of strings in **command-list**; if **argument-list** is **$argv**, the arguments are taken from the command line arguments

The strings in **argument-list** are assigned to **variable** one by one, and the commands in **command-list**, also known as the body of the loop, are executed for every assignment. This process allows execution of the commands in **command-list** as many times as the number of words in **argument-list**. Figure 14.4 depicts the semantics of the `foreach` command.

The following script in the **foreach_demo** file shows use of the `foreach` command with optional arguments. The variable `person` is assigned the strings in **argument-list** one by one each time the value of the variable is echoed, until no strings remain in the list. At this time, control comes out of the `foreach` statement, and the command following `end` is executed. Then, the code following the `foreach` statement (the `exit` statement only in this case) is executed.

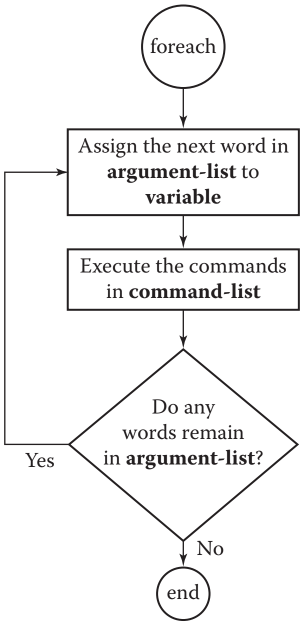

foreach
Assign the next word in **argument-list** to **variable**
Execute the commands in **command-list**
Do any words remain in **argument-list**?
Yes
No
end

FIGURE 14.4 Semantics of the `foreach` statement.

```
% cat foreach_demo
#!/bin/csh
    foreach person ( Debbie Jamie John Kitty Kuhn Shah )
    echo $person
end
exit 0
% ./foreach_demo
Debbie
Jamie
John
Kitty
Kuhn
Shah
%
```

The following script in the **user_info** file takes a list of existing (valid) login names as command line arguments and displays each login name and the full name of the user who owns the login name, one per line. In the sample run, the first value of the user variable is **dheckman**. The `echo` command displays `dheckman:` followed by a `<Tab>`, and the cursor stays at the current line. The `grep` command searches the **/etc/passwd** file for `dheckman` and pipes it to the `cut` command, which displays the fifth field in the **/etc/passwd** line for **dheckman** (his full name). The process is repeated for the remaining two login names (**ghacker** and **sarwar**). As no user is left in the list passed at the command line, control comes out of the `foreach` statement and the `exit 0` command is executed to transfer control back to shell. The command substitution "^"'echo $user":" in the grep command can be replaced by "^"$user":". The >& operator is used to redirect the output and error message generated by the grep command to **/dev/null**, the UNIX black hole. It is so done because the purpose of the grep command is to check whether a login name specified at the command line exists in **/etc/passwd** file or not. The subsequent if statement checks the return status (saved in $?) of the grep command; 0 means success.

```
% cat user_info
#!/bin/csh
foreach user ( $argv )
# Don't display anything if a login name is not in /etc/passwd
   grep "^"'echo $user":"' /etc/passwd >& /dev/null
   if ( $? == 0 ) then
           echo -n "$user :       "
           grep "^"$user":" /etc/passwd | cut -f5 -d':'
   endif
end
exit 0
% ./user_info dheckman ghacker sarwar
dheckman: Dennis R. Heckman
ghacker:  George Hacker
sarwar:   Syed Mansoor Sarwar
%
```

14.7.3 The while Statement

The while statement, also known as the while loop, allows repeated execution of a block of code based on the condition of an expression. The following is a brief description of the statement. Figure 14.5 illustrates the semantics of the while statement.

SYNTAX

```
while (expression)
    command-list
end
```

Purpose: To execute commands in **command-list** so long as **expression** evaluates to true

The **expression** is evaluated and, if the result of this evaluation is true, the commands in **command-list** are executed and **expression** is evaluated again. This sequence of expression evaluation and execution of **command-list**, known as *iteration*, is repeated until the **expression** evaluates to false. At that time, control comes out of the while statement and the statement following end is executed.

The variables and/or conditions in the expression that result in a true value must be properly manipulated in the commands in **command-list** for well-behaved loops—that

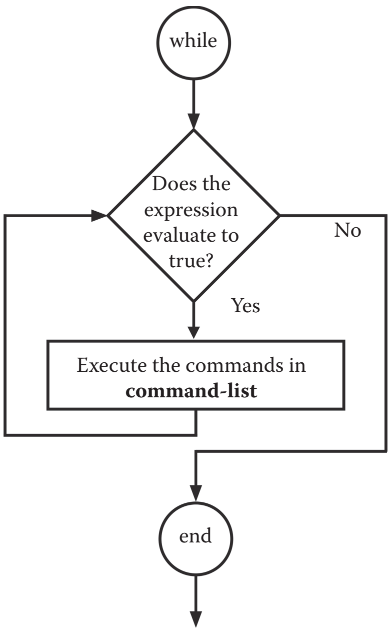

while
Does the expression evaluate to true?
No
Yes
Execute the commands in **command-list**
end

FIGURE 14.5 Semantics of the `while` statement.

is, loops that eventually terminate and allow execution of the rest of the code in a script. Loops in which the expression always evaluates to true are known as *infinite loops*. Infinite loops, also known as *nonterminating loops*, are usually a result of poor design and/or programming, are undesirable because they continuously use CPU time without accomplishing any useful task. However, some applications require infinite loops. For example, all the servers for Internet services, such as the HTTP server, are programs that run indefinitely, waiting for client requests. Once a server has received a client request, it processes it, sends a response to the client, and waits for another client request. The only way to terminate a process with an infinite loop is to kill it by using the `kill` command. Or, if the process is running in the foreground, pressing `<Ctrl+C>` would do the trick, unless the process is designed to ignore `<Ctrl+C>`. In that case, you need to put the process in the background by pressing `<Ctrl+Z>` and using the `kill -9` command to terminate it. We discussed UNIX processes, including the foreground and background processes, in Chapter 10.

The following script in the **while_demo** file shows a simple use of the `while` loop. When you run this script, the `secretcode` variable is initialized to `agent007`, and you are prompted to make a guess. Your guess is stored in the local variable `yourguess`. If your guess is not `agent007`, the condition for the `while` loop is true and the commands between `while` and `end` are executed. The program displays a tactful message informing you of your failure and prompts you for another guess. Your guess is again stored in the `yourguess` variable and the condition for the loop is tested. This process continues until you enter `agent007` as your guess. This time, the condition for the loop becomes false, and control comes out of the `while` statement. The echo command following done executes, congratulating you for being part of a great gene pool!

```
% cat while_demo
#!/bin/csh
set secretcode = agent007
echo "Guess the code!"
echo -n "Enter your guess: "
set yourguess = 'head -1'
while ("$secretcode" != "$yourguess")
    echo Good guess but wrong. Try again!
    echo -n "Enter your guess: "
    set yourguess = 'head -1'
end
echo "Wow! You are a genius!!"
exit 0
% ./while_demo
Guess the code!
Enter your guess: star wars
Good guess but wrong. Try again!
Enter your guess: columbo
Good guess but wrong. Try again!
Enter your guess: agent007
Wow! You are a genius!!
%
```

14.7.4 The break, continue, and goto Commands

The `break` and `continue` commands can be used to interrupt the sequential execution of the loop body. The `break` command transfers control to the command following `end`, thus terminating the loop prematurely. A good programming use of the `break` command is to transfer control out of a nested loop. The `continue` command transfers control to `end`, which results in the evaluation of the loop condition again, hence continuation of the loop. In both cases, the commands in the loop body following these statements are not executed. Thus they are typically part of a conditional statement such as an `if` statement. The `goto` command can be used to transfer control to any location in the script. The following is a brief description of the command.

SYNTAX

```
goto label
```

Purpose: To execute the command at the **label**

The `goto` command transfers control to the command at `label:`, a tag for the command. The use of `goto` is considered a bad programming practice because it makes debugging of programs a daunting task. For this reason, we do not recommend its use, with

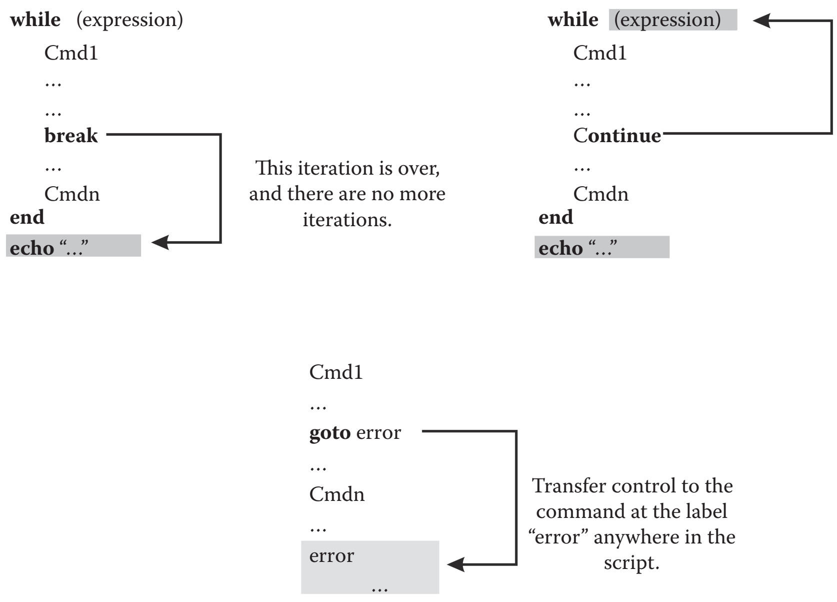

while (expression)
Cmd1
…
…
break
…
Cmdn
end
echo "…"
This iteration is over, and there are no more iterations.
while (expression)
Cmd1
…
…
Continue
…
Cmdn
end
echo "…"
Cmd1
…
goto error
…
Cmdn
…
error
…
Transfer control to the command at the label "error" anywhere in the script.

FIGURE 14.6 Semantics of the `break`, `continue`, and `goto` commands.

the exception perhaps of transferring control out of a nested loop—all loops and not just the one that has the `goto` command in it. Figure 14.6 illustrates the semantics of these commands.

In the following in-chapter exercises, you will write the C shell scripts with loops by using the `foreach` and `while` statements.

EXERCISE 14.10

Write a shell script that takes a list of host names on your network as command line arguments and displays whether the hosts are up or down. Use the `ping` command to display the status of a host and the `foreach` statement to process all host names.

EXERCISE 14.11

Rewrite the script in Exercise 14.10 by using the `while` statement.

14.7.5 The switch Statement

The `switch` statement provides a mechanism for multiway branching similar to a nested `if` statement. However, the structure provided by the `switch` statement is more readable. You should use the `switch` statement when you can—that is, when you are testing a single variable to several distinct patterns. You would not use it when you want to test more than one variable. The following is a brief description of the statement.

SYNTAX

```
switch (test-string)
  case pattern1:
    command-list1
    breaksw
  case pattern2:
    command-list2
    breaksw
  ...
  ...
  default:
    command-listN
    breaksw
endsw
```

Purpose: To implement multiway branching as with a nested `if`

The `switch` statement compares the value in **test-string** with the values of all the patterns one by one until either a match is found or there are no more patterns to match **test-string** with. If a match is found, the commands in the corresponding **command-list** are executed and control goes out of the `switch` statement. If no match is found, control goes to commands in **command-listN**. You don't need to include a default for the `switch` statement. Figure 14.7 illustrates the semantics of the `switch` statement.

The following script in the **switch_demo** file shows a simple but representative use of the `switch` statement. It is a menu-driven program that displays a menu of options

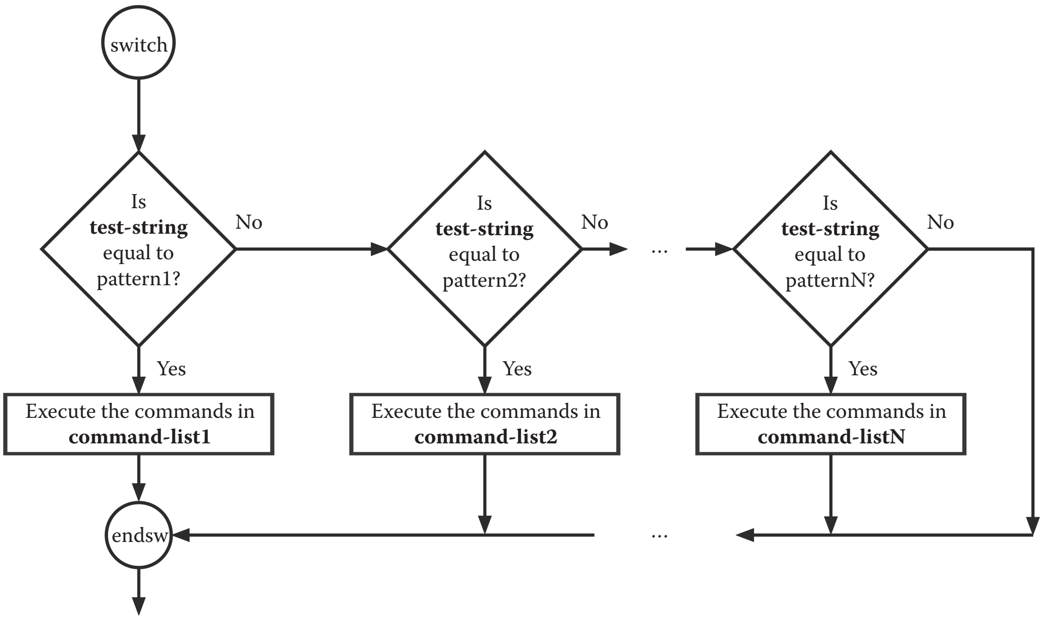

switch
Is test-string equal to pattern1?
No
Yes
Execute the commands in command-list1
Is test-string equal to pattern2?
No
Yes
Execute the commands in command-list2
...
Is test-string equal to patternN?
No
Yes
Execute the commands in command-listN
endsw
...

FIGURE 14.7 Semantics of the `switch` statement.

and prompts you to enter an option. Your input is read into a variable called `option`. The `switch` statement then matches your option with one of the four available patterns (single characters in this case) one by one, and when a match is found, the corresponding **command-list** is executed. Thus if you type `d` and hit `<Enter>` at the prompt, the `date` command is executed and control goes out of `switch`. The `exit 0` command is then executed for normal program termination. Note that the C shell performs logical OR on the items enclosed in brackets. Thus, here, uppercase and lowercase letters are treated the same. A few sample runs of the script follow the code.

```
% cat switch_demo
#!/bin/csh
echo "Use one of the following options:"
echo " d or D: To display today's date and present time"
echo " l or L: To see listing of files in your working directory"
echo " w or W: To see who is logged in"
echo " q or Q: To quit this program"
echo -n "Enter your option and hit <Enter>: "
set option = 'head -1'
switch ("$option")
    case [dD]:
        date
        breaksw
    case [lL]:
        ls
        breaksw
    case [wW]:
        who
        breaksw
    case [qQ]:
        exit 0
        breaksw
    default:
        echo "Invalid option; try running the program again."
        exit 1
        breaksw
endsw
exit 0
% ./switch_demo
Use one of the following options:
 d or D: To display today's date and present time
 l or L: To see listing of files in your working directory
 w or W: To see who is logged in
 q or Q: To quit this program
Enter your option and hit <Enter>: d
Sun Sep 7 11:23:00 PKT 2014
```

```
% ./switch_demo
Use one of the following options:
 d or D: To display today's date and present time
 l or L: To see listing of files in your working directory
 w or W: To see who is logged in
 q or Q: To quit this program
Enter your option and hit <Enter>: L
cmdargs_demo        foreach_demo        shift_demo
display_change_name keyin_demo          switch_demo
display_name        lab3                user_info
export_demo         set_demo            while_demo
% ./switch_demo
Use one of the following options:
 d or D: To display today's date and present time
 l or L: To see listing of files in your working directory
 w or W: To see who is logged in
 q or Q: To quit this program
Enter your option and hit <Enter>: q
% ./switch_demo
Use one of the following options:
 d or D: To display today's date and present time
 l or L: To see listing of files in your working directory
 w or W: To see who is logged in
 q or Q: To quit this program
Enter your option and hit <Enter>: a
Invalid option; try running the program again.
%
```

SUMMARY

Every UNIX shell has a programming language that allows you to write programs for performing tasks that cannot be performed by using the existing commands. These programs are commonly known as shell scripts. In its simplest form, a shell script consists of a list of shell commands that are executed by a shell sequentially, one by one. More advanced scripts use program control flow statements for implementing multiway branching, repetitive execution of a block of commands in the script, transferring control out of a loop, or transferring control to any command in the script. The shell programs that consist of C shell commands, statements, and features are called C shell scripts.

The shell variables are the main memory locations that are given names and can be read from and written to by using these names, instead of the addresses of the relevant memory locations. There are two types of shell variables: environment variables and user-defined variables. Environment variables, initialized by the shell at the time the user logs on, are maintained by the shell to provide a user-friendly work environment. User-defined variables are used as scratch pads in a script to accomplish the task at hand. Some environment variables, such as the positional parameters, are read only in the sense that the user cannot change their values without using the `set` command.

The C shell commands for processing shell variables are `set` and `setenv` (for setting values of positional parameters and displaying values of all environment variables), `env` (for displaying values of all shell variables), `unset` and `unsetenv` (for removing shell variables from the environment set by using the `set` and `setenv` commands, respectively), `set` with `$<` or `'head -1'` (for assigning keyboard input as values of variables), and `shift` (for shifting command line arguments to the left by one or more positions).

The program control flow statements `if` and `switch` allow the user to implement multiway branching, and the `foreach` and `while` statements can be used to implement loops. The `continue`, `break`, and `goto` commands can be used to interrupt the sequential execution of a shell program and transfer control to a statement that (usually) is not the next statement in the program layout.

QUESTIONS AND PROBLEMS

Note: All scripts should be written using the C shell language

1. What is a shell script? Describe three ways of executing a shell script.
2. What is a shell variable? What is a read-only variable? How can you make a user-defined variable read only? Give an example to illustrate your answer.
3. Which shell environment variable is used to store your search path? Change your search path interactively to include the directories **~/bin** and **.** . What would this change allow you to do? Why? If you want to make it a permanent change, what would you do? See Chapter 2 if you have forgotten how to change your search path.
4. What will be the output if the shell script **keyin_demo** in Section 14.3.5 is executed and you give `*` as input each time you are prompted?
5. Write a shell script that takes an ordinary file as an argument and removes the file if its size is zero. Otherwise, the script displays the file's name, size, number of hard links, owner, and modify date (in this order) on one line. Your script must do the appropriate error checking.
6. Write a shell script that takes a directory as a required argument and displays the name of all zero-length files in it. Do the appropriate error checking.
7. Write a shell script that removes all zero-length ordinary files and empty directories in the current directory. Do the appropriate error checking.
8. Modify the script in Problem 6 so that it removes all zero-length ordinary files in the directory passed as an optional argument. If you do not specify the directory argument, the script uses the present working directory as the default argument. Do the appropriate error checking.

9. Run the script in **if_demo2** in Section 14.6 with `if _ demo2` as its argument. Does the output make sense to you? Why?
10. Modify the script in **if_demo2** in Section 14.6 so that it takes two ordinary files as arguments and displays information about both. Also, the script should display an error message if a special file is passed to it as an argument.
11. Write a shell script that takes a list of login names on your computer system as command line arguments and displays these login names and full names of the users who own these logins (as contained in the **/etc/passwd** file), one per line. If a login name is invalid (not found in the **/etc/passwd** file), display the login name but nothing for the full name. The format of the output line is `login name: user name`.
12. What happens when you run a stand-alone command enclosed in back quotes (grave accents), such as `'date'`? Why?
13. What happens when you execute the following sequence of shell commands?
    a. `set name=date`
    b. `$name`
    c. `'$name'`
14. Take a look at your **~/.login** and **~/.cshrc** files and list the environment variables that are exported, along with their values. What is the purpose of each variable?
15. Write a shell script that takes a list of login names as its arguments and displays the number of terminals that each user is logged on to in a LAN environment.
16. Write a shell script `domain2ip` that takes a list of domain names as command line arguments and displays their IP addresses. Use the `nslookup` command. The following is a sample run of this program.

```
% domain2ip up.edu redhat.com
Name: up.edu
Address: 192.220.208.9
Name: redhat.com
Address: 207.175.42.154
%
```

17. Modify the script in the **switch_demo** file in Section 14.6.5 so that it allows you to try any number of options and quits only when you use the `q` option.
18. Write a shell script that displays the following menu and prompts for one-character input to invoke a menu option, as follows:
    a. List all files in the present working directory
    b. Display today’s date and time
    c. Invoke the shell script for Problem 15

d. Display whether a file is a *simple* file or a *directory*

e. Create a backup for a file

f. Start a telnet session

g. Start an ftp session

h. Exit

Option (c) requires that you ask the user for a list of login names. For options (d) and (e), prompt the user for file names before invoking a shell command/program. For options (f) and (g), prompt the user for a domain name (or PI address) before initiating a telnet or ftp session. The program should allow the user to try any option any number of times and should quit only when the user gives option (h) as input. A good programming practice is to build code incrementally—that is, write code for one option, test it, and then go to the next option. Use this style of code development for writing this shell script.

19. Write a shell script that reads a string from the keyboard. If the string is a pathname for a directory in your system's directory structure, the script displays the counts of all types of files in it in the following format:

```
Directories             :
Block special files     :
Character special files :
Link files              :
FIFOs                   :
Ordinary files          :
Sockets                 :
```

The script displays the cumulative size (in bytes) for all ordinary files in the directory. Do the appropriate exception handling.

CHAPTER 15

Advanced C Shell Programming

Objectives

- To discuss numeric data processing
- To describe array processing
- To discuss how standard input of a command in a shell script can be redirected to data within the script
- To explain signal/interrupt processing capability of the C shell
- To cover the commands and primitives

  `=, +=, -=, *=, /=, %=, ==, <, >, |, &, (), <<, @, <^Z>, <^C>, clear, kill, onintr, set, stty`

15.1 INTRODUCTION

We did not discuss four features of C shell programming in Chapter 14: *processing of numeric data*, *array processing*, the *here document*, and *interrupt processing*. In this chapter, we discuss these features and give some example scripts that use them. We also describe how C shell scripts can be debugged.

15.2 NUMERIC DATA PROCESSING

The C shell has a built-in capability for processing numeric data. It allows you to perform arithmetic and logic operations on numeric integer data without explicitly converting string data to numeric data and vice versa. You can use the @ command to declare numeric variables, the variables that contain integer data. This command allows declaration of local variables only. The following is a brief description of the command.

SYNTAX

`@ variable [operator expression] [variable [operator expression]] ...`

Purpose: To declare `variable` to be a numeric variable, evaluate the arithmetic expression, apply the operator specified in `operator` to the current value of `variable` and the value of `expression`, and assign the result to `variable`

Expressions are formed by using the arithmetic and logic operators summarized previously in Table 14.4. Although octal numbers can be used in expressions by starting them with 0, the final value of an arithmetic expression is always expressed in decimal numbers. The elements of an expression must be separated by one or more spaces unless the elements are `(`,`)`, &, |, <, and >. Table 15.1 describes the possible assignment operators that can be used.

In the following interactive session, the numeric variables `value1` and `value2` are initialized to 10 and 15, respectively. The `echo` command is used to show that the assignments work properly.

```
% @ value1=10
% @ value2=15
% echo "$value1 $value2"
10 15
%
```

In the following session, the `@` command declares two variables, `difference` and `sum`, and initializes them to the values of the expressions to the right of the respective = signs. These actions result in the variables `difference` and `sum` taking the values −5 and 25, respectively.

```
% @ difference = ( $value1 - $value2 ) sum = ( $value1 + $value2 )
% echo $difference $sum
-5 25
%
```

TABLE 15.1 Assignment Operators for the @ Command

Operator	Meaning
=	Assigns the value of the expression on the right-hand side of = to the variable preceding it
+=	Adds the value of the expression on the right-hand side of = to the current value of the variable preceding it and assigns the result to the variable
-=	Subtracts the value of the expression on the right-hand side of = from the current value of the variable preceding it and assigns the result to the variable
*=	Multiplies the value of the expression on the right-hand side of = by the current value of the variable preceding it and assigns the result (product) to the variable
/=	Divides the value of the variable preceding = by the value of the expression on the right-hand side of = and assigns the result (quotient) to the variable
%=	Divides the value of the variable preceding = by the value of the expression on the right-hand side of = and assigns the remainder to the variable

You can use the ++ and –– operators, also used in most contemporary high-level languages including C, C++, and Java, to increment or decrement a variable's value by 1. Thus, `@ var1++` and `@ var1––` increase and decrease, respectively, the value of the numeric variable `var1`. Similarly, we can use the following syntax to increment or decrement the value of `var1` by N.

```
@ var1 += N
@ var1 -= N
```

In the following session, we show with examples how you can use these operators.

```
% @ result = $value1 + $value2
% echo $result
25
% @ result++
% echo $result
26
% @ result += 1
% echo $result
27
% @ result = ( $result + 1 )
% echo $result
28
% @ result--
% echo $result
27
% @ result -= 1
% echo $result
26
% @ result -= $value1
% echo $result
16
%
```

You can also use the variables declared the `set` command to store numeric data. Thus, in the following session, the variables `side`, `area`, and `volume` are declared by using the `set` command and are assigned numeric values by using the @ command. The `echo` command displays the values of these variables.

```
% set side = 10 area volume
% @ area = $side * $side
% @ volume = $side * $side * $side
% echo $side $area $volume
10 100 1000
%
```

In the following in-chapter exercise, you will perform numeric data processing by using the `set` and @ commands.

EXERCISE 15.1

Declare two numeric variables `var1` and `var2` initialized to 10 and 30, respectively. Give two versions of a command that will produce and display their sum and product.

15.3 ARRAY PROCESSING

An array is a named collection of items of the same type stored in contiguous memory locations. Array items are numbered according to their locations in the array, with the first item being at location 1. You can access an array item by using the name of the array followed by the item number in brackets. Thus, you can use `people[k]` to refer to the *k*th element in the array called `people`. This process is known as array indexing. You can declare arrays for strings and integers by using the `set` command in the following manner.

SYNTAX

```
set array_name = ( array elements )
```

Purpose: To declare `array _ name` to be an array variable containing "array elements" in parentheses

You can access the contents of the whole array by using the array name preceded by the dollar sign (`$`), such as `$name`. You can access the total number of elements in an array by using the array name preceded by `$#`, as in `$#name`, or simply by using `$#`, as is the case in the Bourne shell. The value of `$?name` is 1 if the named array has been initialized and 0 if it has not been initialized.

In the following session, we define a string array of six items, called `students`, initialized to the strings enclosed in parentheses. The contents of the whole array can be accessed by using `$students`, as shown in the first `echo` command. Thus, the `echo $#students` command displays `6` (the size of the `students` array), and the `echo $?students` command displays `1`, informing you that the array has been initialized. We also show how you can declare an empty array, called `empty`, and initialize an array with the elements of another array; in this case, the `empty` array.

```
% set students = (David James Jeremy Brian Art Charlie)
% echo $students
David James Jeremy Brian Art Charlie
% echo $#students
6
% echo $?students
1
% set empty = ()
% echo $empty
% echo $#empty
0
% echo $?empty
1
```

```
% set students = ( $empty )
% echo $students
%
```

In the following session, we show how elements of the `students` array can be accessed and changed. You can access the *i*th element in the `students` array by indexing it as `$students[i]`. You can display a range of array elements from element at position `i` to the element at position `j` by using `i-j` as the index value, as in `$students[i-j]`. In the session below, the `set` command changes the second element of the `students` array from “James” to “Steve Jobs.” The first `echo` command displays all the elements of the array. The second `echo` command displays the second item in the `students` array. The third `echo` command displays four elements of the `students` array, starting with the item at position 2. The last `set` command, where we try to assign a value to the seventh slot of the `students` array, shows that, once defined, the size of an array is fixed and may not be changed. However, you may redefine an array of a new size by assigning it a new set of values, as shown via the third `set` and fourth `echo` commands in the following session. Finally, you may assign a subarray of an array to another array variable, as shown in the final two `set` and `echo` commands. Figure 15.1 depicts the original and modified `students` arrays.

```
% set students[2] = "Steve Jobs"
% echo $students
David Steve Jobs Jeremy Brian Art Charlie
% echo $students[2]
Steve Jobs
% echo $students[2-5]
Steve Jobs Jeremy Brian Art
% set students[7] = Jamal
set: Subscript out of range.
% set students = ( "Dennis Ritchie" "Ken Thompson" David James
Jeremy "Linus Torvalds" Brian Art Charlie Jamal "Steve Jobs" )
% echo $students
```

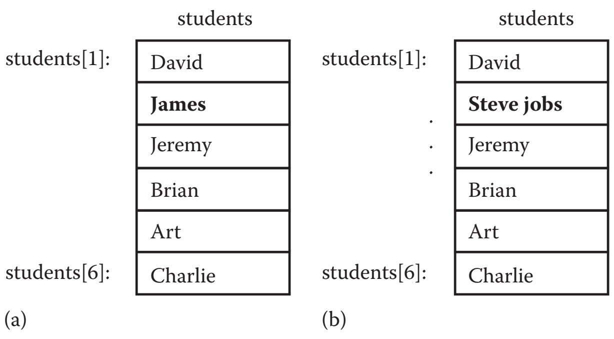


FIGURE 15.1 The `students` array (a) before and (b) after changing the contents of the second slot.

```
Dennis Ritchie Ken Thompson David James Jeremy Linus Torvalds
Brian Art Charlie Jamal Steve Jobs
% set UNIX_Authors = ( $students[1-2] )
% echo $UNIX_Authors
Dennis Ritchie Ken Thompson
%
```

Like other variables, an array variable can be removed from the environment by using the `unset` command. In the following session, the `unset` command is used to deallocate the `students` array. The `echo` command is used to confirm that the array has actually been deallocated.

```
% unset students
% echo $students
students: Undefined variable.
%
```

Any shell variable assigned multiple values with the `set` command becomes an array variable. Thus, when a variable is assigned a multiword output of a command as a value, it becomes an array variable and contains each field of the output, as separated by spaces, in a separate array slot. In the following session, `files` is an array variable whose elements are the names of all the files in the present working directory. The `numfiles` variable contains the number of files in the current directory. The `echo $files[3]` command displays the third array element.

```
% set files = `ls` numfiles = `ls | wc -w`
% echo $files
cmdargs_demo foreach_demo1 if_demo1 if_demo2 keyin_demo
% echo $numfiles
5
% echo $files[3]
if_demo1
%
```

You can also use an array declared with the `set` command to contain numeric data. In the following session, the **num_array_demo** file contains a script that uses an array of integers, called `Fibonacci`, computes the sum of integers in the array, and displays the sum on the screen. The `Fibonacci` array contains the first 10 numbers of the Fibonacci series. If you are not familiar with the Fibonacci series, the first two numbers in the series are 0 and 1 and the next Fibonacci number is calculated by adding the preceding two numbers. Therefore, the first 10 numbers in the Fibonacci series are 0, 1, 1, 2, 3, 5, 8, 13, 21, and 34. Thus, the Fibonacci series may be expressed mathematically as follows.

$$F_1 = 0, F_2 = 1, \ldots, F_n = F_{n-1} + F_{n-2} \quad (\text{for } n \geq 3)$$

The following script in the **num_array_demo** file is well documented and fairly easy to understand. It displays the sum of the first 10 Fibonacci numbers. A sample run of the script follows the code.

```
% cat num_array_demo
# File Name:         ~/unixbook/examples/Cshell/num_array_demo
# Author:            Syed Mansoor Sarwar
# Written:           August 16, 2004
# Last Modified:     August 16, 2004; September 13, 2014
# Purpose:           To demonstrate working with numeric arrays
# Brief Description: Maintain running sum of numbers in a numeric
#                    variable called sum, starting with 0.
#                    Read the next array value and add it to sum.
#                    When all elements have been read, stop and
#                    display the answer.
#!/bin/csh
# Initialize the Fibonacci array to any number of Fibonacci
# numbers - first ten in this case
set Fibonacci = ( 0 1 1 2 3 5 8 13 21 34 )
@ size = $#Fibonacci      # Size of the Fibonacci array
@ index = 1               # Array index initialized to point to
                          # the first element
@ sum = 0                 # Running sum initialized to 0
while ( $index <= $size )
   @ sum = $sum + $Fibonacci[$index]  # Update the running sum
   @ index++                          # Increment array index by 1
end
echo "The sum of the given $#Fibonacci numbers is $sum."
exit 0
% ./num_array_demo
The sum of the given 10 numbers is 88.
%
```

Although this example clearly explains numeric array processing, it is of little or no practical use. We now present a more useful example, wherein the **fs** file contains a script that takes a directory as an optional argument and returns the size (in bytes) of all ordinary files in it. If no directory name is given at the command line, the script uses the current directory. If you specify more than one argument, the script displays an error message and terminates. When executed with one nondirectory argument only, the program again displays an error message and exits. When executed with one argument only and the argument is a directory, the program initializes the string array variable `files` to the names of all the files in the specified directory by using the `set files = 'ls $directory'` command. A numeric variable `nfiles` is initialized to the number of files in the specified directory by using the `@ nfiles = $#files` command. Then, the size of every ordinary file in the `files` array is added to the numeric variable `sum` that is initialized to `0`. When no more names are left in the `files` array, the program displays the total space (in bytes) used by all ordinary files in the directory, then terminates.

```
% cat fs
#!/bin/csh

# File Name:          ~/unix3e/CShell/fs
# Author:             Syed Mansoor Sarwar
# Written:            August 16, 2004
# Modified:           May 11, 2004; September 13, 2014
# Purpose:            To add the sizes of all the files in a
#                     directory passed as command line argument
# Brief Description: Maintain running sum of file sizes in a
#                     numeric variable called sum, starting with 0.
#                     Read all the file names by using the pipeline
#                     of ls, more, and while commands.
#                     Get the size of the next file and add it to
#                     the running sum. Stop when all file names
#                     have been processed and
#                     display the answer.
if ( $# == 0 ) then     # If no command line argument, the
                        # set directory to current directory
    set directory = "."
   else if ( $# != 1 ) then   # If more then one command line
                              # argument then display command
                              # syntax
    echo "Usage: $0 [directory name]"
    exit 1
   else if ( ! -e "$1" ) then # If one command line argument, but
                              # file does not exist, display error
                              # message
        echo "$1 : File does not exist"
        exit 1
   else if ( ! -d "$1" ) then  # If one command line argument, but is
                              # not a directory, show command
                              # syntax
    echo "Usage: $0 [directory name]"
    exit 1
   else
    set directory="$1" # If one command line argument and it is a
                       # directory, prepare to perform the task
endif
# Initialize files array to file names in the specified directory
set files = 'ls $directory'
```

```
@ nfiles = $#files # Number of files in the specified directory
                   # into nfiles
@ index = 1     # Array index initialized to point to the first
                # file name
@ sum = 0       # Running sum initialized to 0
if ( "$nfiles" == 0 ) then
    echo "$directory : Empty directory"
    exit 0
endif

while ( $index <= $nfiles )    # For as long as a file name is
                               # left in files
    set thisfile = $directory/$files[$index]
    if ( -f "$thisfile" ) then    # If the next file is an
                                  # ordinary file
    set argv = 'ls -l $thisfile'  # Set command line arguments
    @ sum = $sum + $argv[5]       # Add file size to the running
                                  # total
    @ index++
    else
    @ index++
    endif
end
echo "The size of all ordinary files in $directory is $sum bytes."
exit 0
% ./fs unix3e
unix3e : File does not exist
% ./fs / /usr/bin
Usage: fs [directory name]
% ./fs dir1
d1 : Empty directory
% ./fs
The size of all ordinary files in . is 7360 bytes.
% ./fs ~
The size of all ordinary files in /home/sarwar is 0 bytes.
% ./fs ~/unix3e
The size of all ordinary files in /home/sarwar/unix3e is 0 bytes.
% ./fs /
The size of all ordinary files in / is 10238 bytes.
% ./fs /usr/bin
The size of all ordinary files in /usr/bin is 232019634 bytes.
%
```

In the following in-chapter exercise, you will write a C shell script that uses the numeric data processing commands for manipulating numeric arrays.

EXERCISE 15.2

Write a C shell script that contains two numeric arrays, `array1` and `array2`, initialized to values in the sets {1, 2, 3, 4, 5} and {1, 4, 9, 16, 25}, respectively. The script produces and displays an array whose elements are the sum of the corresponding elements in the two arrays. Thus the first element of the resultant array is $1 + 1 = 2$, the second element is $2 + 4 = 6$, and so on.

15.4 THE HERE DOCUMENT

The *here document* feature of the C shell allows you to redirect standard input of a command in a script and attach it to data within the script. Obviously, then, this feature works with commands that take input from standard input. The feature is used mainly to display menus, although it also is useful in other ways. The following is a brief description of the here document.

SYNTAX

```
command <<[-] input_marker
… input data …
input_marker
```

Purpose: To execute `command` with its input coming from the here document—data between the input start and end markers "input_marker"

The "input_marker" is a word that you choose to wrap the input data in for `command`. The closing marker must be on a line by itself and cannot be surrounded by any spaces. The command and variable substitutions are performed before the here document data is directed to the standard input of the command. Quotes can be used to prevent these substitutions or to enclose any quotes in the here document. The "input_marker" can be enclosed in quotes to prevent any substitutions in the entire document, as follows:

```
command <<Marker
...
Marker
```

A hyphen (-) after << can be used to remove leading tabs (not spaces) from the lines in the here document and the marker that ends the here document. This feature allows the here document and the delimiting marker to conform to the indentation of the script. One last, but very important point: The output and error redirections of the command that uses the here document must be specified in the command line, not following the here document ending marker. The same is true of connecting the standard output of the command with other commands via a pipeline, as shown in the following session. See Section 13.3 for an example.

We explain the use of the here document with a simple redirection of the **stdin** of the `cat` command from the document. The following script in the **heredoc_demo** file displays a message for the user and then sends a mail message to the person whose e-mail address is passed as a command line argument.

```
% cat heredoc_demo
#!/bin/csh

cat << DataTag
This is a simple use of the here document. This data is the
input to the cat command.
DataTag

# Second example
mail -s "Weekly Meeting Reminder" $argv[1] << WRAPPER

Hello,

This is a reminder for the weekly faculty meeting tomorrow.

Mansoor

WRAPPER

echo "Sending mail to $argv[1] ... done."
exit 0
% ./heredoc_demo ecefaculty
This is a simple use of the here document. This data is the
input to the cat command.
Sending mail to ecefaculty ... done.
%
```

The following script is more useful and a better use of the here document feature. This script, `dext` (directory expert), maintains a directory of names, phone numbers, and e-mail addresses. The script is run with a name as a command line argument and uses the `grep` command to display the directory entry corresponding to the name. The `-i` option is used with the `grep` command to ignore the case of letters.

```
% more dext
#!/bin/csh
if ( $#argv == 0 ) then
    echo "Usage: $0 name"
    exit 1
endif
set user_input = "$argv[1]"
grep -i "$user_input" << DIRECTORY

        John Doe       555.232.0000   johnd@somedomain.com
        Jenny Great    444.6565.1111  jg@new.somecollege.edu
        David Nice     999.111.3333   david_nice@xyz.org
        Don Carr       555.111.3333   dcarr@old.hoggie.edu
        Masood Shah    666.010.9820   shah@Garments.com.pk
        Jim Davis      777.000.9999   davis@great.adviser.edu
        Art Pohm       333.000.8888   art.pohm@great.professor.edu
        David Carr     777.999.2222   dcarr@net.net.gov
DIRECTORY
exit 0
% ./dext
Usage: dext name
% dext Pohm
        Art Pohm       333.000.8888   art.pohm@great.professor.edu
% ./dext Carr
        Don Carr       555.111.3333   dcarr@old.hoggie.edu
        David Carr     777.999.2222   dcarr@net.net.gov
% ./dext david
        David Nice     999.111.3333   david_nice@xyz.org
        David Carr     777.999.2222   dcarr@net.net.gov
%
```

The advantage of maintaining the directory within the script is that it eliminates some extra file operations such as open and read that would be required if the directory data were maintained in a separate file. The result is a much faster program.

Completing the following in-chapter exercise will enhance your understanding of the here document feature of C shell.

EXERCISE 15.3

Create the `dext` script on your system and run it. Try it with as many different inputs as you can think of. Does the script work correctly?

15.5 INTERRUPT (SIGNAL) PROCESSING

We discussed the basic concept of signals in Chapter 10, where we defined them as software interrupts that can be sent to a process. We also stated that the process receiving a signal can take any one of three possible actions:

1. Accept the default action as determined by the UNIX kernel
2. Ignore the signal
3. Take a programmer-defined action

In UNIX, several types of signals can be sent to a process. Some of these signals can be sent via the hardware devices such as the keyboard, but all can be sent via the `kill` command, as discussed in Chapter 10 and Chapter 13. The most common event that causes a hardware interrupt (and a signal) is generated when you press `<Ctrl+C>` and is known as the keyboard interrupt. This event, as its default action, causes the foreground process to terminate. Other events that cause a process to receive a signal include termination of

TABLE 15.2 Some Important Signals, Their Numbers, and Their Purpose

Signal Name	Signal #	Purpose
SIGHUP (hang up)	1	Informs the process when the user who ran the process logs out and terminates the process
SIGINT (keyboard interrupt)	2	Informs the process when the user presses `<Ctrl+C>` and terminates the process
SIGQUIT (quit signal)	3	Informs the process when the user presses `<Ctrl+|>` or `<Ctrl+\>` and terminates the process
SIGKILL (sure kill)	9	Terminates the process when the user sends this signal to it with the `kill -9` command
SIGSEGV (segmentation violation)	11	Terminates the process upon memory fault when a process tries to access memory space that does not belong to it
SIGTERM (software termination)	15	Terminates the process when the kill command is used without any signal number
SIGTSTP (suspend/stop signal)	18	Suspends the process, usually `<Ctrl+Z>`
SIGCHLD (child finished execution)	20	Informs the process of the termination of one of its children

a child process, a process accessing a main memory location that is not part of its address space (the main memory area that the program owns and is allowed to access), and a software termination signal caused by execution of the `kill` command without any signal number. Table 15.2 presents a list of some important signals, their numbers (which can be used to generate those signals with the `kill` command), and their purposes.

The signal processing feature of the C shell allows you to write programs that cannot be terminated by a terminal interrupt (`<Ctrl+C>`). In contrast to the Bourne shell support for signal processing, this feature is very limited. The command used to intercept and ignore `<Ctrl+C>` is `onintr`. The following is a brief description of the command.

SYNTAX

`onintr [options]`

Purpose: To ignore a terminal interrupt (`<Ctrl+C>`) or intercept it and transfer control to any command

Commonly used options/features:

`-`	To ignore the terminal interrupt
`label:`	To transfer control to the command at "label"

When you use the `onintr` command without any option, the default action of process termination takes place when you press `<Ctrl+C>` while the process is running. Thus, using the `onintr` command without any option is redundant. Here, we enhance the script in the **while_demo** file in Chapter 14, so that you cannot terminate execution of this program with `<Ctrl+C>`. The enhanced version is in the **onintr_demo** file, as shown in the following session. Note that the `onintr` command is used to transfer control to the command at the `interrupt_label:` label when you press

`<Ctrl+C>` while executing this program. The code at this label is a `goto` command that transfers control to the `onintr  interrupt` command to reset the interrupt handling capability of the code, effectively ignoring `<Ctrl+C>`. A sample run illustrates this point.

```
% cat onintr_demo
#!/bin/csh

# Intercept <Ctrl+C> and transfer control to the command at
backagain:
    onintr interrupt

# Set the secret code
set secretcode = agent007

# Get user input
echo "Guess the code!"
echo -n "Enter your guess: "
set yourguess = 'head -1'

# As long as the user input is not the secret code (agent007 in
# this case), loop here: display a message and take user input
# gain. When the user input matches the secret code, terminate the
# loop and execute the following echo command.
while ( "$secretcode" != "$yourguess" )
    echo "Good guess but wrong. Try again\!"
    echo -n "Enter your guess: "
    set yourguess = 'head -1'
end
echo "Wow! You are a genius\!"
exit 0
# Code executed when you press <Ctrl+C>
interrupt:
    echo "Nice try -- you cannot terminate me by <Ctrl+C>\!"
    goto backagain
% ./onintr_demo
Guess the code!
Enter your guess: codecracker
Good guess but wrong. Try again!
Enter your guess: <Ctrl+C>
Nice try -- you cannot terminate me by <Ctrl+C>!
Guess the code!
Enter your guess: Lionking
Good guess but wrong. Try again!
Enter your guess: agent007
```

```
Wow! You are a genius!
%
```

The net effect of using the `onintr` command in the preceding script is to ignore a keyboard interrupt. You can achieve the same effect by using the command with the `-` option. Thus, the whole interrupt handling code in the **onintr_demo** program can be replaced by the `onintr -` command; no code is needed at any label, but then the code does not display any message for you when you press <Ctrl+C>.

To terminate programs that ignore terminal interrupts, you have to use the `kill` command. You can do so by suspending the process by pressing <Ctrl+Z>, using the `ps` command to get the process ID (PID) of the process, and terminating it with the `kill` command.

You can modify the script in the **onintr_demo** file so that it ignores the keyboard interrupt, clears the display screen, and turns off the echo. Whatever you enter at the keyboard, then, is not displayed. Next, it prompts you for the code word twice. If you do not enter the same code word both times, it reminds you of your bad short-term memory and quits. If you enter the same code word, it clears the display screen and prompts you to guess the code word again. If you do not enter the original code word, the screen is cleared and you are prompted to guess again. The program does not terminate until you have entered the original code word. When you do enter it, the display screen is cleared, a message is displayed at the top left of the screen, and echo is turned on. Because the terminal interrupt is ignored, you cannot terminate the program by pressing <Ctrl+C>. The `stty -echo` command turns off the echo. Thus, when you type the original code word (or any guesses), it is not displayed on the screen. The `clear` command clears the display screen and locates the cursor at the top-left corner. The `stty echo` command turns on the echo. The resulting script is in the **canleave** file shown in the following session.

```
% cat canleave
#!/bin/csh

# File Name:      ~/unix3e/CShell/canleave
# Author:         Syed Mansoor Sarwar
# Written:        August 18, 2004
# Modified:       May 8, 2004, September 15, 2014
# Purpose:        To allow a user to leave his/her terminal for a
#                 short duration of time by locking the terminal
#                 after taking a code from the user. Terminal is
#                 unlocked only when the user re-enters the same
#                 code. Ignores command line arguments. Does not
#                 terminate with <Ctrl+C>.
# Brief Description:
#                 Clear screen and turn off echo (i.e., do not
#                 display what the user types at the keyboard).
#                 Take user code, save it, and ask the user
#                 to re-enter his/her code just to make sure that
#                 the user remembers the code that he/she has
#                 entered. It is done twice. If the user does not
#                 enter the same code, the program terminates
#                 after displaying a message for the user. The
#                 user is prompted to enter the original code. If
#                 the user enters the wrong code, the program
#                 keeps on prompting the user until he/she enters
#                 the orignal code. The keyboard is then unlocked,
#                 echo is turned on, and program exits, allowing
#                 the user to use his/her system again.
# Ignore terminal interrupt
onintr -

# Clear the screen, locate the cursor at the top-left corner,
# and turn off echo
clear
stty -echo

# Set the secret code
echo -n "Enter your code word: "
set secretcode = `head -1`
echo " "

# To make sure that the user remembers the code word
echo -n "Enter your code word again: "
set same = `head -1`
echo " "
if ( $secretcode != $same ) then
  echo "Work on your short-term memory before using this code\!"
  exit 1
endif

# Keyboard locked. Hit <Enter> to continue.
clear
echo -n "Keyboard locked. Hit <Enter> to continue."
set ignore = `head -1`
clear

# Get user guess to unlock the terminal
echo -n "Enter the code word: "
set yourguess = `head -1`
echo " "

# As long as the user input is not the original code word, loop
# here: display a message and take user input gain. When the user
# input matches the secret code word, terminate the loop and
# execute the following echo command.
```

```
while ( "$secretcode" != "$yourguess" )
    clear
    echo -n "Enter the code word: "
    set yourguess = `head -1`
end

# Set terminal to echo mode
clear
echo "Back again!"
stty echo
exit 0
%
```

You can use this script to lock your terminal before you leave it to pick up a printout or get a can of soda; hence, the name **canleave** (can leave). Using it saves you the time otherwise required for the logout and login procedures.

15.6 DEBUGGING SHELL PROGRAMS

You can debug your C shell scripts by using the `-x` (echo) option of the `csh` command. This option displays each line of the script after variable substitution but before execution. You can combine the `-x` option with the `-v` (verbose) option to display each command line of the script, as it appears in the script file before its execution. You can also invoke the `csh` command from the command line to run the script, or you can make it part of the script, as in `#!/bin/csh -xv`.

In the following session, we show how a shell script can be debugged. The script in the **debug_demo** file prompts you to enter a digit. If you enter a value between 1 and 9, it displays a message informing you that what you entered was good and quits. If you enter any other value, it informs you accordingly and exits. When we execute the script and enter `4`, a valid input, it displays the message `var1cat: Undefined variable.` Similarly, when we run the script and enter `10`, an invalid input, we get the same error message.

```
% cat debug_demo
#!/bin/csh

echo -n "Enter a digit: "
set var1 = `head -1`
if (("$var1" >= 1) && ("$var1" <= 9)) then
   echo "Good input is $var1!!"
else
   echo "Bad input is $var1!!"
endif
exit 0
% ./debug_demo
Enter a digit: 4
```

```
var1cat: Undefined variable.
% ./debug_demo
Enter a digit: 10
var1cat: Undefined variable.
%
```

We debug the program by using the `csh -xv debug_demo` command. The last two lines of output of the runtime trace show the problem area. Somehow, the `$var1` variable is followed by the character sequence `cat canleave`. We put a space between `$var1` and `!!` at the end of the `echo` command. This helps a little in that the output becomes `Good input is 4 cat canleave`. We then realize that the problem is with the two back-to-back bang signs (`!!`) at the end of the two `echo` commands. The bang sign is has a special meaning in the C shell that indicates the beginning of a previous event; `!!` means the event immediately preceding. This means that `cat canleave` was the command that was executed prior to this command. The problem is taken care of by either escaping one of the two `!` signs, as in `$var1\!!`, or by removing one of the `!` signs, as in `$var1!`. After we take care of this problem, the script works properly.

```
% csh -xv debug_demo

echo -n "Enter a digit: "
echo -n Enter a digit:
Enter a digit: set var1 = 'head -1'
set var1 = 'head -1'
head -1
4
if ( ( "$var1" > = 1 ) && ( "$var1" < = 9 ) ) then
if ( ( 4 > = 1 ) && ( 4 < = 9 ) ) then
echo "Good input is $var1cat canleave"
var1cat: Undefined variable.
%
```

For larger scripts, it may become difficult to identify the problem area. In such cases, you should use the `echo` commands at different places in your script to narrow down on the problematic code region. It is similar to using the `cout` or `printf` statements in C, C++, and Java programs while debugging your code in these high-level languages.

The following in-chapter exercise has been designed to enhance your understanding of the interrupt processing and debugging features of the C shell.

EXERCISE 15.4

Test the scripts in the **onintr_demo** and **canleave** files on your UNIX system. Do they work as expected? Be sure that you understand them. If your versions do not work properly, use the `csh -xv` command to debug them.

SUMMARY

The C shell has the built-in capability for numeric integer data processing in terms of arithmetic, logic, and shift operations. Combined with the array processing feature of the language, this allows the programmer to write useful programs for processing large data sets with relative ease. The numeric variables can be declared and processed by using the `@` and `set` commands.

The here document feature of the C shell allows standard input of a command in a script to be attached to data within the script. The use of this feature results in more efficient programs. The reason is that no extra file-related operations, such as file open and read, are needed, as the data is within the script file and has probably been loaded into the main memory when the script was loaded.

The C shell also allows the user to write programs that ignore signals such as terminal interrupt (`<Ctrl+C>`). This useful feature can be used, among other things, to disable program termination when it is in the middle of updating a file. The `onintr` command can be used to invoke this feature.

The C shell programs can be debugged by using the `-x` and `-v` options of the `csh` command, as in `csh -xv filename`. This technique allows viewing of the commands in the user's script after variable substitution but before execution.

QUESTIONS AND PROBLEMS

1. Is the `expr` command needed in the C shell?
2. What is the here document? Why is it useful?
3. Modify the `num_array_demo` script in Section 15.3 so that it takes the numbers to be added as the command line arguments. Use the `while` control structure and integer arrays.
4. What is the difference between the following two commands if `students` is an array? See the second shell session in Section 15.3 for the current value of the `students` array. Explain your answer.

   a. `set UNIX_Authors = $students[1-2]`

   b. `set UNIX_Authors = ( $students[1-2] )`
5. The `dext` script in Section 15.4 takes a single name as a command line argument. Modify this script so that it takes a list of names as command line arguments. Use the `foreach` control structure to implement your solution.
6. The script in the **canleave** file discussed in Section 15.5 is designed to ignore keyboard interrupt. How can this program be terminated? Be precise.
7. Write a C shell script that takes integer numbers as command line arguments and displays their sum on the screen.

8. Write a C shell script that takes an integer number from the keyboard and displays the Fibonacci numbers equal to the number entered from the keyboard. Thus, if the user enters 7, your script displays the first seven Fibonacci numbers.
9. What are signals in UNIX? What types of signals can be intercepted in C shell scripts?
10. Enhance the code of Problem 7 so that it cannot be terminated by pressing `<Ctrl+C>`. When the user presses `<Ctrl+C>`, your script should give a message to the user and continue.
11. Modify the script in Problem 6 so that it reads integers to be added as a here document.
12. Enhance the `onintr _ demo` script so that it takes the code word and the category of the code word (e.g., scientist, TV newscaster, politician, celebrity, movie, sportsperson) as input. It then informs the user of the category and gives 10 chances to the user to guess the code word.
13. Write a C shell script that implements the following menu options:
    a. Display the name of the central processing unit (CPU) used by your system.
    b. Display the name of the operating system used by your computer.
    c. Display the results of a. and b. on the screen separated by a vertical tab.
    d. Display the full pathnames for the commands that have been executed on your system.
    e. Display the maximum number of files a process may open.
    f. Display the maximum number of simultaneous processes a user may have on the system.

    Your program should not terminate when the user presses `<Ctrl+C>`.

    Hint: Review the manual pages for the following commands: `hash`, `ulimit`, `uname`.

CHAPTER 16

Python

Objectives

- To give an overview of the Python programming language
- To cover the basic syntax of Python
- To show how to install and run Python on a UNIX system
- To provide several basic practical examples of using Python in the UNIX environment
- To cover the commands and primitives

  `python`

16.1 INTRODUCTION

In the UNIX environment, if you are presented with a task that requires you to do either script writing or programming, the first thing you have to decide is what programming model you are going to adopt to accomplish the task. In simple terms, this means using any of the three predominant programming models: the procedural/imperative programming model, the object-oriented programming (OOP) model, or the logic programming model. Of course, how well you accomplish the task depends on how much script writing or programming experience you have. And, perhaps, if you are doing this task with a group of people, their experiences and preferences count for a great deal as well. But, how you formulate the task in terms of any of the models is related, most importantly, to your experience in using these models. There are no simple guidelines for applying any of the models to the vast number of possible script writing or programming tasks that exist in UNIX.

However, the bottom line in the accomplishment of your task is how familiar you are with the syntax of the languages that implement the model you choose.

That is why we explain Python syntax in detail. Python can utilize all of these programming models, and can, in fact, mix the techniques used in the models. We make some commentary on this here, and illustrate some simple uses of the models in the next few sections.

Python is a *high-level*, *structured* (in this case, this means built of regular components), and *interpreted* programming (or scripting) language, as opposed to a low-level, compiled language like C or Java. As stated, it is also a *multiparadigm* programming language, which allows you to use data abstractions from the three predominant paradigms. For our purposes, "scripting" and "programming" can be thought of as the same thing.

16.1.1 Python Program Data Model

Even though Python is a multiparadigm programming language, its actual basis is the following:

Everything in a Python program can be referred to as an *object*, with the same exact meaning of the word *object* in OOP. These objects have three parts: an *identifier*, a *type*, and a *value*.

For example, when you assign `X = 62.25` interactively in the Python interpreter, or in a Python module, script file, or library of modules, a real number object type is created; it has a value of 62.25, and it is identified as an object with a pointer to its location in memory. `X` is the identifier that refers to its specific location in memory.

In OOP languages such as Python and Java, the type that an object assumes gives it membership in a particular set, called its *class*. The class of the object limits and also defines, in a fashion, what are known as the *methods*, or operations, which can be performed on or with it.

When a particular object of some type is created, that particular object is called an *instance* of that type. In general, an object's identity and type cannot be changed. They are known as *immutable*. If an object's value can be modified, the object is said to be *mutable*. An object that refers to other objects to obtain value and type is known as a *container*.

Objects can also define their own *attributes* or characteristics of the data they are comprised of, and even the *methods* used on them. An attribute is a property or value associated with an object. A method is a function internal to a class of objects that performs some sort of operation on those objects when the method is invoked.

Attributes and methods are accessed using the dot (`.`) operator, as shown in the following examples:

`x = 2 + 4j` creates a complex number `x`.

`A = a.real` uses a method known as `real` to extract the real part (an attribute) of `a`.

`c = [1, 2, 3]` creates an instance of type list identified as `c` of the integers 1, −2, and −3.

`c.append(7)` adds a new element to `c` using the append method.

16.1.2 The Ultimate Python Reference

Before you begin this section, and as you proceed through the rest of this chapter, it would be helpful for you to read and try to understand, in a "top-down" manner, the following reference in the Python online documentation for the release of Python you are using.

The Python Language Reference: Release 2.7.X, by Guido van Rossum and Fred L. Drake, Jr.

Whatever top-down principles you can carry with you from this reference throughout your Python programming experience, both in this chapter and beyond, are very important and will enable you to see the complex details in a larger context. And we have put a summary of the above reference in Table 16.6, found at the end of this chapter.

16.1.3 Ultimate Reference Glossary

A simplified and abbreviated glossary of some of the terms from this reference is as follows:

Class: A template for creating user-defined objects. Class definitions normally contain method definitions which operate on instances of the class.

Expression: A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators, or function calls, which all return a value.

In contrast to many other languages, not all language constructs are expressions. There are also statements which cannot be used as expressions, such as `print` or `if`. Assignments are also statements, not expressions.

Immutable: An object with a fixed value. Immutable objects include numbers, strings, and tuples. Such objects cannot be altered. A new object has to be created if a different value has to be stored; for example, a key in a dictionary.

Iterable: An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as a list, string, and tuple) and some nonsequence types like `dict` and `file` and objects of any classes you define with an `_ _iter_ _()` or `_ _getitem_ _()` method. Iterables can be used in a `for` loop and in many other places where a sequence is needed. When an iterable object is passed as an argument to the built-in function `iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop.

Lambda: An anonymous inline function consisting of a single expression which is evaluated when the function is called. The syntax to create a lambda function is `lambda [arguments]: expression`.

Method: A function which is defined inside a class body. If called as an attribute of an instance of that class, the method will get the instance object as its first argument (which is usually called self).

mutable: Mutable objects can change their value but keep their class identity.

Pythonic: An idea or piece of code which closely follows the most common usages of the Python language, rather than implementing code using structures common to other languages. For example, a common usage in Python is to loop over all elements of an iterable using a `for` statement. Many other languages do not have this type of construct, so people unfamiliar with Python sometimes use a numerical counter instead:

```
for i in range(len(money)):
    print money[i]
```

The Pythonic way:

```
for bills in money:
    print bills
```

Sequence: An iterable which supports efficient element access using integer indices via the `__getitem__()` special method and defines a `len()` method that returns the length of the sequence. Some built-in sequence types are `list`, `str`, `tuple`, and `unicode`. Note that `dict` also supports `__getitem__()` and `__len__()`, but is considered a mapping rather than a sequence because the lookups use arbitrary immutable keys rather than integers.

type: The kind of object, such as integers or character strings.

16.1.4 Python Standard Type Hierarchy

The *type* of an object describes the Python data structure representation of the object as well as the methods and operations that can be carried out on that object. Table 16.1 is a listing of the type categories, and following it is a brief description of some of the categories in the table.

The `None` type has a single value that contains a null object (an object with no value). Its truth value is `False`.

Numeric types: Booleans, integers, long integers, floating point numbers, and complex numbers.

Sequence types: *Sequences* represent ordered sets of objects indexed by non-negative integers and include strings, Unicode strings, lists, and tuples.

Mapping types: A *mapping object* represents an arbitrary collection of objects that are indexed by another collection of nearly arbitrary key values. Unlike a sequence, a mapping object is unordered and can be indexed by numbers, strings, and other objects. *Dictionaries* are the only built-in mapping type and are similar to hash.

Set types: A *set* is an unordered collection of unique items. Unlike sequences, sets provide no indexing or slicing operations. They are also unlike dictionaries in that there

TABLE 16.1 Python Type Categories

Category		Name	Description
None		`None`	Null object
Numbers		`int`	Plain integer
		`Long`	Arbitrary-precision integer
		`Float`	Floating point number
		`Complex`	Complex number
		`Bool`	Boolean (`True` or `False`)
Sequences (immutable)		`str`	Character string
		`Unicode`	Unicode character string
		`tuple`	Tuple
Sequences (mutable)		`list`	List
		`bytearray`	Returned by `bytearray()`
Mapping		`dict`	Dictionary
Sets		`set`	Mutable set
		`Frozenset`	Immutable set
Callable		`BuiltinFunctionType`	Built-in functions
		`BuiltinMethodType`	Built-in methods
		`type`	Type of built-in types and classes
		`object`	Ancestor of all types and classes
		`FunctionType`	User-defined function
		`InstanceType`	Class object instance
		`MethodType`	Bound class method
		`UnboundMethodType`	Unbound class method
"	Modules	`ModuleType`	Module
"	Classes	`object`	Ancestor of all types and classes
"	Types	`type`	Type of built-in types and classes
"	Files	`file`	File
"	Internal	`CodeType`	Byte-compiled code
		`FrameType`	Execution frame
		`GeneratorType`	Generator object
		`TracebackType`	Stacks traceback of an exception
		`Slice`	Generated by extended slices
		`Ellipsis`	Used in extended slices
"	Classic	`Classes ClassType`	Legacy class definition
		`InstanceType`	Legacy class instance

are no key values associated with the objects. In addition, the items placed into a set must be immutable.

Callable types: These represent objects that support the function call operation. There are several flavors of objects with this property, including user-defined functions, built-in functions, instance methods, and classes.

Classes and types: When you define a class, the class definition normally produces an object of type `type`.

Modules: The module type is a container that holds objects loaded with the `import` statement.

Files: The *file* object represents an open file and is returned by the built-in `open()` function.

Internal types: Objects used by the interpreter are exposed to the user, such as *traceback objects*, *code objects*, *frame objects*, *generator objects*, *slice objects*, and the *ellipsis object*.

Code objects: These represent raw byte-compiled executable code, or *bytecode*, and are typically returned by the built-in `compile()` function.

Frame objects: These are used to represent execution frames and most frequently occur in traceback objects.

Traceback objects: These are created when an exception occurs and contains *stack trace information*.

Generator objects: These are created when a generator function is invoked (see Chapter 6, "Functions and Functional Programming"). A generator function is defined whenever a function makes use of the special `yield` keyword.

Slice objects: These are used to represent slices given in extended slice syntax, such as `a[i:j:stride]`, `a[i:j, n:m]`, or `a[..., i:j]`.

Ellipsis object: The ellipsis object is used to indicate the presence of an ellipsis (...) in a slice. There is a single object of this type, accessed through the built-in name `Ellipsis`. It has no attributes and evaluates as `True`.

Classic classes: In versions of Python prior to version 2.2, classes and objects were implemented using an entirely different mechanism that is now deprecated. For backward compatibility, however, these classes, called *classic classes* or *old-style classes*, are still supported.

16.1.5 Basic Assumptions We Make

Four basic and important assumptions we make in this chapter are

1. You have Python 2.X installed on your UNIX system. This installation was either done by the system administrator at the time the UNIX system was installed on the computer you are using, or by you. On our base UNIX system, PC-BSD (the one we have used to illustrate *everything* in this book), Python 2.7.9 was already installed as part of the installation of the system itself.

2. The path of execution to the Python program and the path of execution to all the Python scripts you create in this chapter include the current working directory that

you want to do Python in! If you do not know, given the particular shell you are using (we use the C shell, with the `%` prompt), what your path of execution is set to, go back to Section 2.8 and examine your path and set it properly. For example, in the C shell, you can see your path of execution by typing **`echo $path`** at the shell prompt. On our base UNIX system, PC-BSD, Python is installed by default in **/usr/local/bin**.

3. You are doing Python in a console or terminal window *without* an integrated development environment (IDE). Therefore, the basic procedure is as follows: you edit Python scripts in your favorite text editor, save them to the current working directory, and execute Python in that current working directory. You execute Python interactively by typing commands into its "interpreter" shell window. When you become more familiar with Python, you may wish to make your work with the language more efficient by using an IDE.
4. In general, whenever we want you to type something on the Python command line, we will indicate what is to be typed in **bold** text.

Computer programs execute and accomplish their objectives in a particular order, from start to finish. They may "branch" within that order, perhaps to only execute some of their instructions, based on certain logical tests or conditions. They may also repeat segments of their operation, either for some predetermined number of times, or indeterminately, based on changing conditions. Python conforms to this model, and operates using the following scheme of levels:

1. *Everything* in Python programs, or scripts, is composed of modular components.
2. These modular components contain syntactically correct Python statements.
3. These statements contain expressions.
4. The expressions create and manipulate objects.

16.1.6 Running Python

The following subsection illustrates the three ways that we use to run Python.

16.1.6.1 Way 1 (Interactive Mode)

In a console or terminal window, you type **`python`**. The program executes, and you are presented with the Python command prompt, >>>. Then, you type single or multiple lines of Python code on the Python command line, and see the results immediately. A good reason to use this mode is that you can test small fragments of Python code one line at a time this way directly in the interpreter. A simple example of this would be as follows:

Example 16.0.a

```
>>>print "How about some more?"
How about some more?
>>>
```

To submit a line of Python code to its interpreter, at the end of the line, press `<Enter>` on the keyboard.

16.1.6.2 Way 2 (Script Mode)

You use a text editor of your choice to create and save multiple properly formatted Python commands in a file, called a *script file*, perhaps named **first.py,** in the current working directory in which you are executing Python. Then, you run Python with the **`python`** command and **`first.py`** as the command argument. A good reason to use this mode is if you have scripts with more than a few lines of code in them, and you do not want to type that code in every time you want to run it. A simple example of this would be as follows (the `%` is the C shell command prompt):

Example 16.0.b

```
%python first.py
```

where **first.py** is a file full of syntactically correct Python commands in the current working directory, and the Python program is in the path of execution of programs in your UNIX environment.

This method of executing the Python code is sometimes called running it as a user-written *library* module.

If the commands in **first.py** do not contain any output directed to the screen, such as using print statements, the shell prompt will immediately reappear, and you will not be in the Python interpreter when the script file terminates!

16.1.6.3 Way 3 (Import Script Mode)

Similarly to Way 2, you use a text editor of your choice to create and save multiple Python commands in a script file, perhaps named **first.py**, in the current working directory in which you are executing Python. Then, you run Python, and at the Python command prompt you bring the script file into Python with the **`import`** command. A good reason to use this mode is if your script files contain function definitions. A simple example of this would be as follows:

Example 16.0.c

```
>>>import first
>>>
```

where **first** is the file without the **.py** extension. It should contain Python commands and be in the current working directory. Now, the objects, statements, expressions, and modules (like Python functions) in **first.py** are available to you in the Python interpreter. A good reason to use this mode is to bring those structures and functions into the current interactive Python session.

Caution: Once you leave Python by holding down the `<Ctrl>` and D keys on the keyboard, the current interactive session is ended, and the environment you have created in Python is lost.

16.1.7 Uses of Python

Python can accomplish several kinds of programming tasks, which might be broken down into the following sample categories:

Shell scripting
Systems programming
Network and Internet scripting
Database programming
Systems administration scripting
Graphical user interface (GUI) scripting
Scientific and math programming
Data mining

In this chapter, we use all three modes of running Python shown, and we give a beginners' introduction to the Python language. We also follow the model of level schemes shown, going roughly from the bottom of the scheme to the top. The remainder of this chapter can be divided into three parts:

Section 16.2 shows you how to install Python.
Section 16.3 gives the basic syntax of Python within that model scheme.
Section 16.4 gives a few simple, worked examples of some of the practical programming task categories for which you can use Python in UNIX.

We use Python 2.7.9, a more established and universal Python release at the time of the writing of this book, that comes bundled with and is automatically installed on our base UNIX system, PC-BSD. An advantage of running Python 2.X is that there are many existing libraries of Python 2.X programs and modules that you can incorporate into your own programs. Where necessary, we will note the differences between Python Release 2.X and Python Release 3.X, which for the beginning programmer are not difficult to overcome.

16.2 HOW TO INSTALL PYTHON ON A PC-BSD AND SOLARIS SYSTEM

Both PC-BSD and Solaris come with some version of Python already installed and usable by a normal user. But, you may need to install Python on your system if you are doing the install of some other UNIX operating system and it does not include Python, or if you want to install a later version of the software alongside or to replace the version your system already has on it. There are four possible paths through this section. Be aware that because of variations in the way your system has been installed by the system administrator, and exactly what version of UNIX you have installed, the four-step installation procedure in the next section may have to be done by a system administrator for you! A good example of a similar situation would be that you do not have a C compiler available in **/usr/bin**, or you want to upgrade to the latest *gcc* compiler and do not know how to do that with or without a package manager!

16.2.1 Installing Python on PC-BSD

The easiest way of knowing whether or not Python version 2.7.X is already installed on the computer you are using is to simply type **python** and press <Enter> at the shell prompt in a console or terminal window. When we do this on our PC-BSD system we obtain the following:

```
[bob@pcbsd-6064] ~% python
Python 2.7.9 (default, Feb 12 2014, 19:30:28)
[GCC 4.2.1 Compatible FreeBSD Clang 3.3 (tags/RELEASE_33/final
183502)] on freebsd10
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

The three greater-than symbols (>>>) are the Python interpreter prompt, letting you know you are in Python!

To exit to the command line prompt in the terminal window, press <Ctrl-D> after the >>>.

The first line of response from the system shows that Python 2.7.9 is running on this system.

If you get an error message on the command line in the terminal window, such as command not found, either Python has not been installed on your system, or you do not have access to it. You would need to then contact your system administrator to install the program or give you privileges to execute it.

Every program in this chapter can be done in Python 2.7.9, but if you want to run the examples in Section 16.4.4, you *must* obtain Python 2.7.9 (or the latest release of version 2.7 at the time you are reading this) and install it on your system.

The easiest way of installing Python on a PC-BSD system is to use the command **pkg install python** as superuser, and follow the prompts the **pkg** program provides. For example, if you want to install Python 2.7.X or Python 3.X, type **pkg install python27** or **pkg install python3**. If you do install Python 3.X, to run it on the command line type **python3.4**.

16.2.2 Installing Python on Solaris

For Solaris, you can upgrade the release of Python that comes preinstalled with the system by using the IPS package manager. If you have the default Gnome desktop running under Solaris, there is a graphical version of the IPS package manager that we recommend you use to download and install the latest release of Python 2.7.X. See Figure 16.22 at the end of this chapter for an illustration of the graphical IPS package manager as it appears in Solaris. We do not provide installation instructions to use the IPS package manager graphically in this chapter, but in Chapter 23, "UNIX System Administration Fundamentals," we do go over the details of using the command line, text-based version of the IPS package manager for Solaris to install packages.

16.3 BASIC SETUP AND SYNTAX, AND GETTING HELP

To use Python in practical examples, as shown in later sections of this chapter, it is first necessary to become familiar with the syntax of the language; for example, how it is used as a calculator to execute single lines of Python code to accomplish short, meaningful tasks.

If you need help on a particular module, keyword, or topic in Python, you can always type in the function call to help as follows to get into the help system:

```
>>>help ( )
Welcome to Python 2.7! This is the online help utility.
If this is your first time using Python, you should definitely
check out
the tutorial on the Internet at http://docs.python.org/tutorial/.
Enter the name of any module, keyword, or topic to get help on
writing
Python programs and using Python modules. To quit this help
utility and
return to the interpreter, just type "quit".
To get a list of available modules, keywords, or topics, type
"modules",
"keywords", or "topics". Each module also comes with a one-line
summary
of what it does; to list the modules whose summaries contain a
given word
such as "spam", type "modules spam".
help>
```

16.3.1 Printing Text, Comments, Numbers, Grouping Operators, and Expressions

One of the first things you must know about how a calculator works is how to enter numbers and mathematical expressions on the calculator. Instead of listing all the syntactic rules, we will do a number of examples to illustrate and have you work with those rules.

Here are a few important considerations you need to be aware of before you enter any Python code on the Python command line or into a file.

1. The rule of four: The *indentation* spaces that you place on each line of Python code are very important! Since Python is a structured programming language that uses specific structures in blocks, those blocks are delimited or defined by the indentation you give them on each line (unlike in other languages that use specific printing characters to delimit blocks). This means you must line up your blocks of Python structures vertically, starting from the left-hand side, and for our purposes, use four

spaces for each indented block. For example, the following sample shows this four-space indentation constraint:

```
Block 1 head
    xxxxxxxxx
    xxxxxxxxx
    Block 2 head
        xxxxxxxxx
        xxxxxxxxx
        Block 3 head
            xxxxxxxxx
            xxxxxxxxx
        end of Block 3
    end of Block 2
end of Block 1
```

where Blocks 1, 2, 3 and so on and their statements `xxxxxxxxx` line up vertically with an indentation of four spaces for each block from left to right. This is shown in more detail next.

2. Normal order and applicative order evaluation: The order of execution of a mathematical expression used by Python is PEMDAS: parentheses, exponents, multiplication, division, addition, and subtraction. See Table 16.2 for an even more detailed of operator precedence in Python expressions.

TABLE 16.2 Python Order of Evaluations

Operator	Description
`()`	Parentheses (grouping)
`f(args...)`	Function call
`x[index:index]`	Slicing
`x[index]`	Subscription
`x.attribute`	Attribute reference
`**`	Exponentiation
`~x`	Bitwise not
`+x, -x`	Positive, negative
`*, /, %`	Multiplication, division, remainder
`+, -`	Addition, subtraction
`<<, >>`	Bitwise shifts
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR
`in, not in, is, is not, <, <=, >, >=, <>, !=, ==`	Comparisons, membership, identity
`not x`	Boolean NOT
`and`	Boolean AND
`or`	Boolean OR
`lambda`	Lambda expression

The first thing we want Python to do is to print, or echo, a line of text we type at the keyboard. This is done by typing the following at the Python 2.7 command prompt:

```
>>>print "This is the number of fingers I am holding up:"
This is the number of fingers I am holding up:
```

In Python 3.X, the print statement turned into a function, so the syntax for the previous example in Python 3.X would be:

```
>>>print("This is the number of fingers I am holding up:")
This is the number of fingers I am holding up:
```

To add comments to a script file of Python commands, you place the pound sign(#) before everything on the line you want commented. For example, in interactive mode:

```
>>> # This is a comment, which you can use to annotate your script
file code.
…
>>> # Anything after the # is ignored by python.
…
>>>print "You could have comments appear like this:" # and the
comment after is ignored.
You could have comments appear like this:
>>> # You can also use the pound sign to comment out a piece of
code:
…
>>> # print "This won't run."
…
>>> print "This will run."
This will run.
>>>
```

In this interactive session, just press `<Enter>` on the Python command line when the … appears.

Quotation marks are used for string literals. To include a single quotation mark in a string literal, enclose it inside of double quotation marks. For example:

```
>>> '"Don\'t," he said.'
'"Don\'t," he said.'
>>> print('"Don\'t," he said.')
"Don't," he said.
>>> s = 'First place.\nSecond place.'          # \n means newline
>>> s                                          # without print(),
\n is included in the output
'First line.\nSecond line.'
```

```
>>> print(s)                                   # with print(), \n
produces a new line
First place.
Second place.
```

Triple quotation marks are used to enclose long lines of string literals.

Next, we want to combine some text with an arithmetic expression that Python evaluates for us:

```
>>> print "Not Ring Fingers", 7 – (1 + 1)
Not Ring Fingers 5
```

Notice that in evaluating the mathematical expression, Python evaluates what is in parentheses first by doing the addition of 1 + 1, then, from left to right, subtracts 2 from 7.

EXERCISE 16.1

Have Python evaluate the following expressions, and list what Python prints:

1. `((7 + 5) * (3 + 2))/(6/18)`
2. How can you change the previous expression so that it yields a numeric answer, and why?
3. `3 + 2 +1– 5 + 4 % 2–1 / 4 + 6`
4. What kind of operator is the percent sign (`%`)?

We can also use relational operators in arithmetic expressions, such as less than (<), greater than (>), greater than or equal to (>=), and less than or equal to (<=).

For example:

```
>>> print "Is it true that 3 + 1 < 5 - 7?"
Is it true that 3 + 1 < 5 - 7?
>>> print 3 + 1 < 5 - 7
False
>>> print "Is it greater?", 4 > -2
Is it greater? True
>>> print "Is it greater or equal?", 4 >= -2
Is it greater or equal? True
>>> print "Is it less or equal?", 4 <= -2
Is it less or equal? False
```

EXERCISE 16.2

1. What are the results of typing in the following Python statements, and why?

```
>>> 5 + 7>= 6 <= 78 – 9
>>> 5 + 7>= 6 >= 78 – 9
```

```
    >>> (5 + 7>= 6 >= 78 - 9)/8
    >>> (5 + 7>= 6 >= 78 - 9)/-8
```

16.3.2 Variables

An important feature of a high-level programming language like Python is providing for names that allow you to refer to computational objects. The name represents, or stands for in any particular computational environment of interest, the value or values which the object can take on; thus it is called a 'variable'.

Python variable names can contain both letters, numbers, and the underscore (_) character, but must begin with a letter. If you get an error message about the use of a variable name, it may be a reserved word, or keyword, in Python. Table 16.3 lists the 31 keywords that may not be used as variable names in a Python statement.

In Python 3.X, `exec` is no longer a keyword, but `nonlocal` is.

In the following simple example, we define some variables, and use them:

```
>>> pie=3.14159
>>> radius=10
>>> pie*(radius*radius)
314.159 >>> circumference=2*pie*radius
>>> circumference
62.8318
```

16.3.3 Functions

Python provides a programming construct called a *function* that allows you to define your own named procedures and lets you reuse those procedures in a modular fashion in your code. You can think of a function as a black box machine that takes named objects present before the function call or invocation as inputs, processes them inside the black box with names that are only seen inside the black box, and then spits them out as named objects available to the Python program via an assignment statement.

The general form of a function definition is:

```
def name (formal parameters):
       body of the function
       return (returned parameters)
```

TABLE 16.3 Python 2.7.X Keywords

and	del	from	not	while
as	elif	global	or	with
assert	else	if	pass	yield
break	except	import	print	
class	exec	in	raise	
continue	finally	is	return	
def	for	lambda	try	

where name is the name you give the function (make sure you are not using a Python keyword!), "formal parameters" are the named objects that are passed to the function so it can carry out some operations on them and, optionally, "returned parameters" are named objects that are used by your program.

There are three basic ways you can execute a function:

1. Use Way 1 (Interactive mode), and by typing or copying and pasting each line of code that is the function definition into the interpreter at the Python command line prompt.
2. Use Way 2 (Script mode), and create the function definition in a file with a **.py** extension. Then run Python at the shell prompt with the name of the function definition file plus extension as an argument to the Python command.
3. Use Way 3 (Import script mode), and by importing a file (sometimes called a *module*), which you have created with a text editor and which contains the function definition, into the current session of Python. You can then use the function definitions and named objects in that file.

A simple example of the first way to use a function definition in Python is as follows. Type the following seven lines of Python code at the Python command prompt. The ellipses (…) are another form or the Python prompt, requesting more command line entry. Remember to indent the lines shown, and press <Enter> on the keyboard after each).

```
>>> w = 7
>>> x = 12
>>> def add(a, b):
…          c = a + b
…          return (c)
…
>>> z = add(w, x)
>>> z
19
>>>
```

From this example, we can call w and x the actual parameters passed to the function definition, a and b the formal parameters used in the body of the function definition, and c a returned parameter. Notice that, for the returned parameter to be used in the remainder of your Python session after the function definition, you must assign a named variable on the left-hand side of the equals sign to the function call or invocation on the right-hand side of the equals sign.

A simple example of using Way 3 (Import script mode) to bring a function definition into Python is as follows:

Use the text editor of your choice to create the following file, named **math1.py**, in the current working directory you are executing Python in:

Example 16.1

```
def add(a, b):
    c = a + b
    return(c)
def subtract(a, b):
    c = b - a
    return(c)
def multiply(a, b):
    c = a * b
    return(c)
def divide(a, b):
    c = a / b
    return(c)
```

Then, at the Python command line, type the following three lines of Python code:

```
>>> import math1
>>> z = math1.add(3, 4)
>>> z
7
>>>
```

The first line you typed in at the Python command line made the named objects in the **math1.py** module available in the current session of Python. The second line allowed you to address the add function from that module as `math1.add`. The assignment statement on the second line also allowed you to add 3 and 4 together and assign the returned value to a variable named `z`. We will cover more about Python modules, and global and local scopes in functions, in Section 16.3.16.

16.3.4 Conditional Execution

As mentioned in Section 16.1, the order in which computer programs execute includes a "branching", or conditional execution structure. Python uses the truth value of test conditions to determine whether certain blocks of code will be executed or not. This is implemented in Python with the `if` statement. The general form of the `if` statement construct is

```
if cond1 is true:
    initial statement(s)
elif cond2 is true:
    additional statement(s)
else:
    final statement(s)
```

where:

`cond1` is a test condition whose truth value determines whether the initial statement(s) block gets executed

`cond2` is a test condition whose truth value determines whether the additional statement(s) block gets executed
`else` is the default which executes final statement(s)

A simple example of the use of this structure using Way 1 (Interactive mode) in Python is as follows (remember to press `<Enter>` on the keyboard after each line you type in).

Example 16.2

```
>>> x = 1
>>> if x == 0: # the == is a logical, or Boolean operator
         print "x equal 0"
    elif x == 1: #one or more of these optional blocks are
                 # allowed
         print "x equal 1"
    else:        #optional block
         print "x is something else"
x equal 1
>>>
```

EXERCISE 16.3

1. If you leave out the `elif` block in Example 16.2, what prints out?
2. If you change the first line to read `x = 3`, and leave out the `else` line in Example 16.2, what prints out?
3. If you change the first line to read `x = 1.8` in Example 16.2, what prints out?

It is also possible to nest conditional execution blocks inside of one another. For example:

Example 16.3

```
>>> w = 36
>>> y = 13
>>> z = 20
>>> if w < 37:
...     print "w is less than 37"
...     if y > 13:
...          print "y is greater than 13"
...     elif y == 13:
...          print "y is equal to 13"
...     else:
...          print "y is less than 13"
...     if z > 21:
...          print "z is greater than 21"
...     elif z == 21:
```

```
...         print "z is equal to 21"
...     else:
...         print "z is less than 21"
... else:
...     print "w is greater than or equal to 37"
...
w is less than 37
y is equal to 13
z is less than 21
>>>
```

16.3.5 Determinate and Indeterminate Repetition Structures and Recursion

Python can repeat segments of program or script file structure in two basic ways: via *determinate repetition structures* or *indeterminate repetition structures*. Traditionally, a determinate repetition structure is called *counting repetition*, and an indeterminate repetition structure is called *logical repetition*. These two methods are implemented with the `for` procedural statement and the `while` procedural statement. Generally, if you know ahead of time (at the time you are writing the code) how many repetitions of a block of code you want to execute, you use the `for` statement, and if you do not (when, for example, you allow the user to input the number of repetitions as the script file is run), you use the `while` statement.

Of course, it is possible to implement the same two ways of repetition by *not* using a specific structured programming approach; for example, by using conditional execution and unstructured switching to obtain the same results. But, we choose to show the structured approach in Python.

Make sure that in the body of statements included in the indeterminate repetition block that `while` is executing, the test condition for continued execution becomes false. Otherwise, this will result in *infinite repetition*! To halt infinite repetition in an executing script file, hold down the `<Ctrl>` and `C` keys on the keyboard.

Proceeding from Guido van Rossum's definition of *Pythonic* in the Python language reference, a common technique in Python is to loop over all elements of an iterable object using a `for` statement. Many other languages do not have this type of construct, so people unfamiliar with Python sometimes use a numerical counter instead:

```
>>>for i in range(len(money)):
...     print money[i]
```

The Pythonic way is:

```
>>>for bills in money:
...     print bills
```

The Pythonic way can be characterized as *iteration*, whereas the traditional language construct can be characterized as *counting*.

The for structure can repeat a block of operations on any iterable sequence, such as strings, lists, tuples, or user-defined iterable objects in classes.

The general form of the for statement structure is:

```
for a certain number of times
    repeat these statements
```

The while structure can repeat a block of operations as long as a test condition is true.

The general form of the while statement structure is:

```
while a certain condition is true
    repeat these statements
```

An object, K, is iterable if it can be successfully run with the following code. This code also shows that a counting form of iteration such as the traditional for loop structure can be implemented with a while structure (be careful of indentation, and press <Enter> after the last ellipsis [...]):

```
>>>K = [22,33,44,55]                #Lists are iterable.
>>>c = K.__iter__()                 #c is the counter
>>>while 1:                         #execute while true
...    try:
...        item = c.next()          #get the next one
...        print item               # Do some operations on each item
                                    as you count through it
...    except StopIteration:        #Nothing left
...        break
...

22
33
44
55
>>>
```

Following are two simple examples of both forms of repetition. We use Way 2 (Script mode) to run them:

Save the following three lines of Python in a file named **for1.py** in the current working directory (notice that the keywords in the Python script file are in **bold** type):

Example 16.4

```
limit = [1, 2, 3, 4, 5]
for number in limit:
    print "number of repeats %d" % number
```

Then execute Python on your shell command line with the command **`python for1.py`.** Your output should be:

```
number of repeats 1
number of repeats 2
number of repeats 3
number of repeats 4
number of repeats 5
```

Save the following seven lines of Python in a file named **while1.py** in the current working directory. (Notice that the keywords in the Python script file are shown in **bold** type.)

Example 16.5

```
s= int(raw_input("Enter an Integer"))
i = 0
numbers = []
while i < s:
    numbers.append(i)
    i = i + 1
    print "numbers now: ", numbers
```

Then execute Python on your shell command line with the command **`python while1.py`**. When prompted for an integer, type in `6` and then press `<Enter>` on the keyboard. Your output should be

```
numbers now: [0]
numbers now: [0, 1]
numbers now: [0, 1, 2]
numbers now: [0, 1, 2, 3]
numbers now: [0, 1, 2, 3, 4]
numbers now: [0, 1, 2, 3, 4, 5]
```

Try this same script file using different input integers each time you run it to confirm that indeterminate repetition is happening.

Another interesting indeterminate repetition method that Python can implement is known as *recursion*. Basically, recursion is the repetition of a body of calculations to accumulate intermediate results, until some basic state is reached, and at that point the calculations yield the final results. The following is an example of a recursive process that calculates the factorial of an integer, implemented in Python as a function that calls, or invokes, itself an indeterminate number of times (the Python keywords are shown in **bold** type, and we are using Way 3 [Import script mode] to execute the script):

Example 16.6

```
def factR(n):
    if n == 1:
        return n
    else:
        return n*factR(n - 1)
```

If you create this function definition in a file named **FactR.py** in the current working directory in which you are executing Python, then typing the following into the Python interpreter will yield the factorial of the input argument:

```
>>> import FactR
>>> FactR.factR(7)
5040
>>>
```

EXERCISE 16.4

1. What error message do you get if you supply a real number, such as 9.76, when you run the code of Example 16.6? Why do you get this error message?
2. How can you find the factorial of a real number in Python? Such as from 1.?

16.3.6 File Input and Output

If you look back to the beginning of this chapter, at the programming tasks that Python is capable of in a UNIX environment (such as systems programming, network and Internet scripting, database programming, systems administration scripting, GUI scripting, scientific statistical math programming, and data mining), the common thread which runs through all of those tasks is the ability to interface with the UNIX system via utilities that work with UNIX files. For example, you may be programming a statistical analysis script in Python that perhaps has its data generated from some other program or utility stored in a file somewhere in the file structure of your system. These files can be either text (and in the case of Python 3.X, Unicode text), or binary, raw 8-bit bytes. Probably the most significant differences between Python 2.X and Python 3.X lies in the area of text representation and processing!

The general form of a file operation is

```
name = open(filename, mode)
name.method(argument(s))
name.close()
```

where:

`name` is a file object name in the current procedure
`open` is the keyword that opens Python's connection to an external file

`filename` is the name of the external file, which may include directory paths, and so on
`mode` is a method of accessing the file, like reading from it, or writing to it
`method` is one of several operations that can be performed on the open external file
`argument(s)` are one or more qualifiers on the operation specified in method
`close` is the termination of connection to the external file

The mode can be `r`, `w`, or `a`, for reading (the default, meaning you do not need to specify this to open with read), writing, or appending to the file, respectively. The file will be created if it does not exist, and opened for writing or appending. It will be truncated when opened for writing. Add `b` to the mode for binary files. Add + to the mode to allow simultaneous reading and writing.

The preferred way to open a file is with the built-in `open()` function. Add `U` to the mode to open the file for input with universal newline support.

In the following examples, we illustrate some simple operations on files, such as how to open, write/read from, and close files.

In the first example, we input a string of text into a named file. Run Python and type the following three lines of code into the interpreter:

Example 16.7

```
>>> file = open('sometext', 'w')
>>> file.write('This is a line of text.')
>>> file.close()
>>>
```

Then, when you are in the same working directory in which you are executing Python, type the following line at the UNIX shell prompt (shown as `%`):

```
% more sometext
This is a line of text.
%
```

In the next example, we input some integer data into an external file using your favorite text editor, and then do some Python operations on that data. Run your favorite text editor and into a file you name **somedata.txt**, type the following four integers (one integer per line):

```
23
33
43
54
```

Then, run Python and execute the following lines of code. The variable named `x1` is a *list*, which we will discuss in more detail in Section 16.3.7. So, the list element x1[0] is the first element of the list that has been read from the first line in the external file. The **float** and **int** functions convert the text strings in the file to integers and real numbers:

Example 16.8

```
>>> file = open('somedata.txt')   #the default mode is reading
>>> x1 = file.readlines()
>>> x1
['23\n', '33\n', '43\n', '54\n']
>>> s = float(x1[0])
>>> s
23.0
>>> r = int(x1[0])
>>> r
23
>>> s + r
46.0
>>> file.close()
>>>
```

We can also write list elements, such as numbers, to an external file. For example, the following Python code uses the write method to place three lists into a file named **listw.txt**:

Example 16.9

```
>>> L = [1, 2, 3]
>>> M = [4, 5, 6]
>>> N = [7, 8, 9]
>>> F = open('listw.txt','w')
>>> F.write(str(L) + 'n')
>>> F.write(str(M) + 'n')
>>> F.write(str(N) + 'n')
>>> F.close()
>>>
```

On the shell commend line, view the contents of **listw.txt**:

```
%more listw.txt
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]
%
```

EXERCISE 16.5

1. What commands do you use to add the first three elements of the list x1 of Example 16.8, and print that sum?

2. What are the \n characters shown on line 4 of Example 16.8?
3. What happens if you edit the file **somedata.txt**, and enter the numbers 23, 33, 43, and 54 on a single line in your text editor instead of on four different lines, save the file, and try to perform the same action in Example 16.8 in Python on the new file **somedata.txt**?
4. After you close the file, can you still access the values 43 and 54 in any way?
5. How would you specify the third element of the list x1?

16.3.7 Lists and the List Function

On the lowest level of the organizational scheme for Python, the *list* is an object that can contain multiple elements of possibly different types. Just like a shopping list can contain different kinds of things from a store, such as food, household goods, automotive supplies, and so on, a Python list can be made up of different types of elements, like integers, real numbers, strings, and so on; in fact, any type of Python object. For example:

```
>>>A = [ 34, 'Bob', 54.76, [4,7,9]]
```

is an expression that assigns the integer 34, the string "Bob", the real number 54.76, and another list comprised of the numbers 4, 7, and 9 to a variable named A. List indices are integers, starting with 0. So, the following statements in Python yield the results:

```
>>>A[2]
54.76
>>>A[0]
34
>>>A[3][2]
9
```

A 2×2 matrix (or array) can be specified as:

```
B = [[x, x],[y, y]]
```

Here is Python code to create a 3×3 matrix (or array) named x of random numbers, using the list function that works on objects (please notice that your results will differ from the output shown here, since the numbers shown in the example are randomly generated):

Example 16.10

```
>>> import random # random is a function from the Standard
Library
>>> x = list(list (random.random() for i in range(3)) for j in
range(3))
>>> x
```

```
[[0.1455440585876967, 0.7525092872509719,
0.30168961326498955], [0.6967960669374997, 0.8715621457012694,
0.24960628313623423], [0.891389814359208, 0.9591605275600708,
0.5240885874508074]]
>>>
```

16.3.8 Strings, String Formatting Conversions, and Sequence Operations

16.3.8.1 Strings

Strings are a class of objects in Python that can represent text, and are basically seen in their single-quoted and double-quoted form, which are interchangeable. For example:

```
>>> 'program', "program's"
('program',"program's")
>>>
```

To format strings in an expression, you can use the `%` binary operator to format values as strings according to a specific format definition. On a line of code, on the left of the `%` operator, put in a format string that has one or more code types. On the right of the `%` sign, put in objects you want to substitute in for the types.

The operator (*s* `%` *k*) produces a formatted string, given a format string *s* and a collection of objects in a tuple or mapping object (dictionary).The string *s* may be a standard or Unicode string. The format string contains two types of objects: ordinary characters (which are left unmodified) and conversion specifiers, each of which is replaced with a formatted string representing an element of the associated tuple or mapping.

If *k* is a tuple, the number of conversion specifiers must exactly match the number of objects in *k*. If *k* is a mapping, each conversion specifier must be associated with a valid key name in the mapping, using parentheses. Each conversion specifier starts with the `%` character and ends with one of the conversion characters shown in Table 16.4.

TABLE 16.4 String Formatting Conversions

Character	Output Format
`d, I`	Decimal integer or long integer
`u`	Unsigned integer or long integer
`o`	Octal integer or long integer
`x`	Hexadecimal integer or long integer
`X`	Hexadecimal integer (uppercase letters)
`f`	Floating point as `[-]m.dddddd`
`e`	Floating point as `[-]m.dddddde±xx`
`E`	Floating point as `[-]m.ddddddE±xx`
`g, G`	Use `%e` or `%E` for exponents less than −4 or greater than the precision
`s`	String or any object. The formatting code uses `str()` to generate strings
`r`	Produces the same string as produced by `repr()`
`c`	Single character
`%`	Literal `%`

The following example allows you to perform some basic operations on strings, such as *concatenating* them (adding their characters together), embedding escape sequences in them (to include special characters), finding their lengths (an integer representing their length), or *slicing* them (extracting smaller substring parts of them).

Example 16.11

```
>>> a = 'programming'
>>> b = 'programmer\n'
>>> c = 'programs'
>>> print (a + ' ' + b + c)
programming programmer
programs
>>> len(a+b+c)                    #len is the length operator
30
>>> d = a[0:3] + b[3:7] + c[7] #b[3:7] is gram
>>> d
'programs'
>>> b[3:]
'grammer\n'
>>> print b[3:]
grammer
>>> q = c[:]
>>> q
'programs'
>>>
```

16.3.8.2 Sequence Operations

Three important and useful operations you can perform on sequence types of objects are *indexing*, *slicing*, and *extended slicing*. Objects such as strings and tuples are immutable and cannot be modified after creation. But, lists can be modified with the following operators (Table 16.5).

The following section describes and gives examples of some sequence operations on mutable objects.

TABLE 16.5 Indexing, Slicing, and Extended Slicing Operations

Operation	**Description**
`s[n]`	Returns *n*th element of s
`s[i] = x`	Index assignment
`s[i:j] = r`	Slice assignment
`s[i:j:stride] = r`	Extended slice assignment
`del s[i]`	Deletes an element
`del s[i:j]`	Deletes a slice
`del s[i:j:stride]`	Deletes an extended slice

Indexing into the sequence:

Gets components using offsets, where the first element indexed is at zero (0) offset.

Negative indices count backward from the end, where the last element is at offset –1.

`s[0]` gets the first element, `s[1]` gets the second element, and so on.

`s[–2]` gets the second from last element.

Slicing the sequence:

Extracts contiguous sections of a sequence, from `i` to `j-i`.

Slice boundaries `i` and `j` default to 0 and sequence length `len(s)`.

`s[1:4]` retrieves elements from offset 1–3.

`s[1:]` retrieves from offset 1 until the end of the sequence object.

`s[:–1]` retrieves from offset 0 to the next to last element.

`s[:]` makes a copy of the sequence object.

Extended slicing of the sequence:
The third element is a *stride*, which defaults to 1, added to the offset of each element extracted.

`s[::2]` is every other item in the sequence.

`s[::–1]` is the reverse of the sequence.

`s[4:1:–1]`retrieves from offset 4, up to but not including 1, in reverse.

Slice assignments:

On mutable objects, deleting elements of the sequence and then reinserting new ones.

Iterable objects assigned to slices `s[i:j]` do not have to be the same size.

Iterable objects assigned to extended slices `s[i:j:k]` must match in size.

Here are several interactive code examples of sequence object operations on a list of integers:

```
>>>m = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>n = m[::2]
>>>m
[2, 6, 10, 14, 18 ]
>>>p = m[::-2]
>>>p
```

```
[20, 16, 12, 8, 4]
>>>q = m[0:5:2]
>>>q
[2, 6, 10]
>>>r = m[5:0:-2]
>>>r
[12, 8, 4]
>>>s = m[:5:1]
>>>s
[2, 4, 6, 8, 10]
>>>t = m[:5:-1]
>>>t
[20, 18, 16, 14]
>>>u = m[6::1]
>>>u
[14, 16, 18, 20]
>>>v = m[5::-1]
>>>v
[12, 10, 8, 6, 4, 2]
>>>w = m[5:0:-1]
>>>w
[12, 10, 8, 6, 4]
>>>
```

Here are interactive code examples of some further uses of formatting expressions on different objects.

Example 16.12

```
>>>x = 400
>>>y = 75.142783
>>>z = "master"
>>>d = {'x':13, 'y':1.54321, 'z':'unive'}
>>>q = 1234567812345678L
>>>print 'x is %d' % x
x is 400
>>>print '%10d %f' % (x,y)
400 75.142783
>>>print '%+010d %E' % (x,y)
+0000000400 7.514278E+01
>>>print '%(x)-10d %(y)0.3g' % d
13 1.54
>>>print '%0.4s %s' % (z, d['z'])
mast unive
>>>print '%*.*f' % (5,3,y)
75.143
>>>print 'q = %d' % q
q = 1234567812345678
```

Here are more interactive code examples showing slicing operations on a list of integers.

```
>>>x = [1,2,3,4,5]
>>>x[1] = 6
>>>x
[1,6,3,4,5]
>>>x[2:4] = [10,11]
>>>x
[1,6,10,11,5]
>>>x[3:4] = [-1,-2,-3]
>>>x
[1,6,10,-1,-2,-3,5]
>>>x[2:] = [0]
>>>x
[1, 6, 0]
>>>
```

A slicing assignment may be supplied with an optional stride argument. The argument on the right side of the assignment statement must have exactly the same number of elements as the slice that is being replaced. Here are a few interactive code examples of this.

```
>>>y = [1, 2, 3, 4, 5]
>>>y[1::2] = [10, 11]
>>>y
[1, 10, 3, 11, 5]
>>>y[1::2] = [30, 40, 50]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: attempt to assign sequence of size 3 to extended
slice of size 2
>>>
```

EXERCISE 16.6

1. Instead of printing the concatenated strings in line 4 of Example 16.11, what do you get echoed to you on the Python command line if you just type **`a + b + c`**?
2. The variable `c` has 12 characters in it. Why are there 30 characters returned for the length of the concatenation of `a`, `b`, and `c`?
3. What is the first index value used to extract substrings from a string?
4. What is the last index value used to extract substrings from a string?
5. What does the indexing operation `[:]` shown in Example 16.11 accomplish?
6. What does `a[1:3]` return, and why?

16.3.9 Tuples

Similar to the list object described in Section 16.3.7, the Python *tuple* is a simple object that creates data structures using any other object type. Tuples support most of the same operations as lists, such as indexing, slicing, and concatenation. But, you cannot modify the contents of a tuple after creation, as you can with a list.

This is its most important feature as a sequence object used in data structures; that it cannot be changed. The following example uses Way 2 (Script mode) to execute the code, and shows the creation, querying, and manipulation of tuples in a database.

Example 16.13

With your favorite text editor, create the following file of six lines (named **data.txt**) in the current working directory in which you run Python in:

```
Conditions,23,45.8
Methods,12,11.75
Objects,40,17.1
Modules,1023,1.4
Dictionaries,45,120.71
Comprehensions,5,234.75
```

Next, using your text editor again, create the following file (named **searcher.py**) in the current working directory:

```
filename = "data.txt"
collection=[]
for line in open(filename):
    fields = line.split(",")        #splits line by commas
    name = fields[0]                #create the fields
    uses = int(fields[1])
    value = float(fields[2])
    card = (name,uses,value)        #create the tuple
    collection.append(card)
print collection[0]
print collection[3][2]
sum = 0.0
for name, uses, value in collection:
    sum += uses / value
print sum
```

Then, at the shell prompt, type **`python searcher.py`**
You should see the following output on your screen:

```
('Conditions', 23, 45.8)
1.4
734.9710205576091
>>>
```

16.3.10 Sets

Another elementary object type in Python is the *set*, which is an *unordered* collection of objects that have no duplicate elements. The distinguishing feature of a set as an unordered sequence is that you cannot address or index into the set using the index operations on sequences we illustrated for lists and tuples. To create sets, and do some operations on them, do the following example. (You can omit typing in the comments shown!)

Example 16.14

```
>>> x = set([1.0, 2.0, 3.0, 4.0])
>>> x
set([1.0, 2.0, 3.0, 4.0])
>>> z = set([10,11,12,13,14])
>>> z
set([10, 11, 12, 13, 14])
>>> q = set("Hello")
>>> q
set(['H', 'e', 'l', 'o'])       #no repeated elements
>>> a = z | q                   #union of z and q
>>> a
set(['e', 'H', 10, 11, 12, 13, 14, 'l', 'o'])  #ordered
                                               ascending
>>> r = set([9, 8, 7, 6])
>>> b = q | r
>>> b
set(['e', 6, 7, 8, 'H', 'l', 'o', 9])  #ordered ascending even
                                       though r is descending!
>>> c = r | a
>>> c
set(['e', 6, 7, 8, 9, 10, 11, 12, 13, 14, 'l', 'o', 'H'])
>>> c = r & a                    #intersection of r and a
>>> c
set([])                          #the empty set
>>> x.add(5.0)                   #add an element
>>> x
set([1.0, 2.0, 3.0, 4.0, 5.0])
>>> x.remove(4.0)                #remove an element
>>> x
set([1.0, 2.0, 3.0, 5.0])
>>> x.update([6.0, 7.0, 8.0])    #add multiple elements
>>> x
set([1.0, 2.0, 3.0, 5.0, 6.0, 7.0, 8.0])
>>>
```

EXERCISE 16.7

1. In Example 16.14, why is c finally the empty set?
2. What would the set x contain if you added three number 5.0's with only one x.add command? What would the set x contain if you added three number 5.0's with the x.update command?
3. Can you use slice assignment statements to reassign new values to elements of a tuple?

16.3.11 Dictionaries

A *dictionary* is a data structure or "container" that acts like a table of objects that can be indexed into, or various parts of it can be addressed by, keys. The keys can be strings or one of several other Python objects. The container is not a sequence object, in the way that a list is. An example of dictionary creation that uses strings as keys is as follows.

```
>>> function = {
...              "name":"operator",
...              "class" :"arithmetic",
...              "number": 12
...             }
>>> function
{'number': 12, 'name': 'operator', 'class': 'arithmetic'}
>>>
```

The keys are the strings "name", "class", and "number", the field data are "operator", "arithmetic", and 12, and the curly braces are the syntax that allows you to define the dictionary.

The order of how the keys and their field data is presented is not necessarily the same order in which you defined them!

Here is a way of extracting a value from the table function we just created, and then changing one of the data values in it:

```
>>> n = function.get("number")
>>> n
12
>>> function["class"]="logical"
>>> function
{'number': 12, 'name': 'operator', 'class': 'logical'}
>>>
```

Here is a way that lets you sort the keys in a for loop using the sorted function. Python has two basic ways of sorting. The sorted function works to sort any iterable object, such as entries in a Python dictionary. The sort method is a list method that works on Python lists (there is no need to type in the comments).

```
>>> K = {'x':1, 'y':2, 'z':3}      #creates the dictionary
>>> K
{'y':2, 'x':1, 'z':3}              #not a sequence, thus comes
                                   back in any order
>>>for key in sorted(K):
...    print key, '=', K[key]      #press Enter twice here
...
x = 1
y = 2
z = 3
>>>
```

16.3.12 Generators

A Python functional technique of program execution that harnesses the advanced programming methodologies of data flows, streams, and process pipelines, and preserves the state of the generation of output as the function executes stepwise, is called a *generator*. A generator produces a collection of output results only when a next method (or a next() in Python 3.X) is called. The next method is executed by the Python yield statement. When a large collection of data needs to be created on the fly, perhaps in single steps, at a particular time during program execution, the generator function is invoked. The next built-in function steps you through and generates the output. The following example shows how to create and invoke a generator function (no need to type in the comments):

Example 16.15

```
>>> def generadd(Q):
...     for i in range(Q):
...         yield i        #generates the next value
...         i += 1
...
>>> for i in generadd(4):  #whenever the function is called,
                           the values are generated
...     print i
...
0
1
2
3
>>> z = generadd(6)        #this passes 6 to generadd
>>> z                      #this will show you the compiled
                           generator object
<generator object generated at 0x284f60f4>
>>> next(z)                #the next built-in steps you
                           through and generates the value
```

```
0
>>> next(z)
1
>>> next(z)
2
>>> next(z)
3
>>> next(z)
4
>>> next(z)
5
>>> next(z)
Traceback(most recent call last):
 File "<stdin?", line 1. in <module>
StopIteration
>>>
```

The following example shows how to implement recursive generator functions in a file named **regen.py** that you create with your favorite text editor. First, create the file **regen.py** as shown, and then use Way3 (Import script mode) to do in-chapter Exercise 16.8:

Example 16.16

```
def abc():
    a = deff()
    for i in a:
        yield i
    yield 'abc'
def deff():
    a = ijk()
    for i in a:
        yield i
    yield 'deff'
def ijk():
    for i in (1, 2, 3):
        yield i
    yield 'ijk'
```

EXERCISE 16.8

1. Give the exact Python code that would bring the three functions from Example 16.16 into the Python interpreter, given that you must use Way 3 (Import script mode).
2. Give the exact Python code that would invoke the three functions from Example 16.16 on the Python command line.

3. Give the exact Python code that would allow you to step through the invocation of the three functions from Example 16.16, to generate its output results until you reach `StopIteration`. List the output generated at each step through the recursion.

16.3.13 Coroutines

In the previous section, we introduced generator functions, which use `yield` to give output results. Python generator functions can also "consume" results using a `yield` statement. In addition, two new methods applied to generator objects, `send()` and `close()`, create a framework for objects that consume and give values. Generator functions that define these objects are called *coroutines*. Coroutines consume values using a `yield` statement on the right side of an expression, as follows.

```
value = (yield)
```

With this syntax, execution pauses at this statement until the object's send method is invoked with an argument:

```
coroutine.send(data)
```

Then, execution resumes, with value being assigned to the value of data. To signal the end of a computation, we shut down a coroutine using the `close()` method. This raises a `GeneratorExit` exception inside the coroutine, which we can catch with a `try`/`except` clause.

The next example illustrates these concepts. It is a coroutine that prints strings that match a provided template pattern:

Example 16.17

```
>>>def grepper(template):
...    print ('Searching for ' + template)
...    try:
...        while True:
...        x = (yield)
...        if template in x:
...            print x
...    except GeneratorExit:
...        print "Done"
...
>>> q = grepper("Pythonista")
>>> q.next()
Pythonista
>>> q.send("After doing this section, you will be known as a
Pythonista")
After doing this section, you will be known as a Pythonista
>>> q.send("Not a very Pythonic answer")
>>> q.send("Python makes C look high maintenance and too
complex")
>>> q.close()
Done
>>>
```

When we call q.send a value, evaluation resumes inside the coroutine q at the statement line= (yield), where the sent value is assigned to the variable line. Evaluation continues inside q, printing out the line if it matches, going through the loop until it encounters the line= (yield) again. Then, evaluation pauses inside q and resumes where q.send was called.

We can chain functions that send() and functions that yield together to achieve complex behaviors, similar to streaming or pipelining, illustrated in earlier chapters in this book on shell programming. In the next example, the function read splits a string named text into words and sends each word to another coroutine.

Example 16.18

```
>>> def read(text, next_coroutine):
...     for line in text.split():
...         next_coroutine.send(line)
...     next_coroutine.close()
```

Each word is sent to the coroutine bound to next _ coroutine, causing next _ coroutine to start executing, and this function to pause and wait. It waits until next _ coroutine pauses, at which point the function resumes by sending the next word or exiting with Done.

If we join this function together in a pipeline with the function grepper defined in Example 16.17, we can create a program that prints out only the words that match a particular word.

```
>>> text = "0110 1100 0101 1000 1010 0111 1111 0001 0110"
>>> found = grepper('01')
>>> found.next()
Looking for 01
>>> read(text, found)
0110
0101
1010
0111
0001
0110
Done
>>>
```

The read function sends each word to the coroutine grepper, which prints out any input that matches its template pattern. Within the grepper coroutine, the line `x = (yield)` waits for each word sent, and it transfers control back to `read` when it is reached.

EXERCISE 16.9

1. (a) Put the code from Example 16.17 into a file (if you have not done so already), and using Way 3 (Import script mode), invoke the coroutine `grepper` on the template "`1201`".

   (b) Then search the following using that template: "`0120 1201 1020`", "`3012 3013 3212`", "`12010203101213012`".

   (c) What prints out on your screen when you use the proper commands, similarly to what is shown in the follow-up code to the function definition in Example 16.17?

2. (a) Put the code from Example 16.18 in a file (if you have not done so already), and using Way 3 (Import script mode), invoke the coroutine `read` on the string "`Python is the most Pythonic enterprise a Pythonista can practice`".

   (b) Then search for the string "`Python`" using the coroutines `grepper` and `read`.

   (c) What prints out on your screen when you use the proper commands, similarly to what is shown in the follow-up code to the function definition in Example 16.18?

16.3.14 Objects and Classes

OOP is a programming model that represents concepts as *objects* that have fields (attributes that describe the object) and associated procedures known as *methods*. Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs. Some examples of OOP languages are: Smalltalk, C++, C#, Java. Perl, Ruby, PHP, and Python.

Some common terms used in OOP are as follows:

Class: A user-defined model for an object that defines characteristics of any object in that class. The characteristics are data members (class variables and instance variables) and methods, accessed via dot notation.

Class variable: A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods.

Data member: A class variable or instance variable that holds data associated with a class and its objects.

Instance variable: A variable that is defined inside a method and belongs only to the current instance of a class.

Inheritance: The transfer of the characteristics of a class to other classes that are derived from it.

Instance: An individual object of a certain class. An object that belongs to a class `Circle`, for example, is an instance of the class `Circle`.

Instantiation: The creation of an instance of a class.

Method: A special kind of function that is defined in a class definition.

Object: A unique instance of a data structure defined by its class. An object can comprise both data members (class variables and instance variables) and methods.

The general form of a class definition is:

```
class ClassName:
     'Optional class documentation string'
      class_suite
```

The class has a documentation string, which can be accessed via `ClassName. _ _ doc _ _ .`

The `class _ suite` consists of all the component statements defining class members, data attributes, and methods.

The following lines of interactive code very simply illustrate the inheritance model of OOP classes and its hierarchic nature. Be sure to press `<Enter>` on the second line of code. There is no need to type in the comment line numbers.

```
>>> class tab: pass                                   #Line 1
...                                                   #Line 2
>>> tab.name  = 'Mansoor Sarwar'                      #Line 3
>>> tab.age  = 25                                     #Line 4
>>> print tab.name,tab.age                            #Line 5
Mansoor Sarwar 25
>>> x = tab()                                         #Line 6
>>> y = tab()                                         #Line 7
>>> x.name                                            #Line 8
Mansoor Sarwar
>>> y.name = 'Alan Turing'                            #Line 9
>>> tab.name, x.name, y.name                          #Line 10
('Mansoor Sarwar', 'Mansoor Sarwar', 'Alan Turing')
>>>
```

A line-by-line analysis and description of this code is as follows.

Line 1: Starting with the keyword `class`, you name it `tab`. You use the keyword `pass` to assign the class to an empty namespace object, that is, it has no class members, attributes or methods yet. A class is an object!

Line 2: Continue by pressing `<Enter>`.

Line 3: You now add an attribute called `name` to the class `tab`. The class `tab` has no instances yet!

Line 4: You assign another attribute called `age` to the class `tab`.

Line 5: You print out the attributes of `tab`.

Line 6: You now assign an instance, named `x`, to the class `tab`, which is an empty instance.

Line 7: You now assign another instance, named `y`, to the class `tab`, which is another empty instance.

Line 8: The instance `x` inherits the attribute `name` from `tab`. You use the dot (.) operator to connect or refer to the instance `x` with `name` "Mansoor Sarwar" in the class `tab`.

Line 9: You now explicitly assign the instance `y`, with an attribute `name`, the value "Alan Turing". You use the dot (.) operator to connect the instance `y` with `name` "Alan Turing".

Line 10: You print out the name in `tab`, the name referred to in `x` inherited from `tab`, and the explicitly assigned name in `y`. Attribute references work through the mechanisms of inheritance, and attribute assignments work on the objects to which the assignment is done.

The following is a more involved example of creating a class, and then using some methods to manipulate the objects in that class. Type the following code into a file named **first class.py** using your favorite text editor:

Example 16.19

```
#!/usr/local/bin/python
class Structure:
     'Common base class for all Python Structures'
     StrucCount = 0
     def __init__(s, name, number):
         s.name = name
         s.number= number
         Structure.StrucCount += 1

     def displayCount(s):
         print "Total Structures %d" % Structure.StrucCount
     def displayStructure(s):
         print "Name : ", s.name,  ", Number : ", s.number
```

Then, at the UNIX shell prompt, run Python with the command **`python firstclass.py`**

On the Python command line, type the following (you can leave out the comments):

```
>>> import firstclass
>>> Stru1 = firstclass.Structure("Arithmetic Operators", 17)
#creates the first object
>>> Stru2 = firstclass.Structure("Logical Operators", 10)
#creates the second object
>>> Stru1.displayStructure()     #displays the first object
Name : Arithmetic Operators,, Number : 17
>>> Stru2.displayStructure()     #displays the second object
Name : Logical Operators,, Number : 10
>>> print "Total Structures %d" % firstclass.Structure.
StrucCount #prints the total
Total Structures 2
>>> Stru1.inst = 7           #creates a new attribute of Stru1
>>> hasattr(Stru1, 'inst')   #checks object for attribute
True
>>> getattr(Stru1, 'inst')   #gets the value of the attribute
7
>>> getattr(Stru1, 'name')   #gets the value of the attribute
'Arithmetic Operators'
```

There are three things to notice about this example.

1. The variable `StrucCount` is a class variable whose value is shared among all instances of this class. This variable can be accessed as `Structure.StrucCount` from inside the class or as `firstclass.Structure.StrucCount` outside the class.

2. The first class method, `_ _ init _ _ ()`, is a special method, which is called a *class constructor* or *initialization* method. Python calls this method when you create a new instance of this class.

3. You declare other class methods like normal Python functions, with the exception that the first argument to each method is `s`. Python adds the `s` argument to the list for you; you do not need to include it when you call the methods.

16.3.15 Exceptions

Before we begin our discussion of Python exceptions, it is worth noting that there are facilities that can help you to debug your program 1) before you even submit it to the interpreter, and 2) during the execution of the program. Usually these facilities are a part of a UNIX Python integrated development environment (IDE), which we have *not* been using in this chapter to keep our Python tutorial here as plain and universal with respect to our base UNIX system (PC-BSD) as possible.

A good example of one of these facilities is automatic indentation, available in a Python IDE editor. Another is an interactive step-by-step debugging tool such as PyDebug.

With that said, an exception, or unexpected end to a program, is a Python object that represents an error. To terminate the flow of execution of a program because of some exception the interpreter has found, Python has two kinds of exception-handling structure that can end the program. These structures are

1. Exception handling that uses for example `try:…except:…else:` as shown in Example 16.15, and the standard exceptions—for example, `StopIteration`, as shown in Example 16.13.
2. Assertions, for example using the `assert` statement.

   The general form of `try:...except:...else:` is:

```
try:
    Some operations…
    ...
except ExceptionI:
    If there is an Exception, do this…
else:
    If there is no Exception then, do this…
```

The following is a simple exception test example, which opens a new file in the current working directory, writes some content to the file, and then exits normally.

Example 16.20

```
>>> try:
...     handler = open("datafile", "w")
...     handler.write("This is a data file for testing
        exception handling!!")
... except IOError:
...     print "Error: Can\'t find file or write data"
... else:
...     print "File write successful"
...     handler.close()
...
File write successful
>>>
```

EXERCISE 16.10

1. List five other standard exceptions, what general class of exception they signal, and what source you used to obtain their names.
2. Modify the code of Example 16.18 so that it writes the integers 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10 as a string list to the file named **datafile**.

16.3.16 Modules, Global and Local Scope in Functions

At a certain level of abstraction, everything in Python is a module, even Python itself. A Python module is a container or package which holds all the hierarchical objects, statements, expressions, and other modular components we spoke about in the introduction to this chapter that are necessary to accomplish an intended task or subtask. Modules may contain function definitions, class operations, and variable assignments. Basically, there are only two types of Python module: a module written by you, or a module from some external library, like the Python standard library. The standard library that is built in contains over 200 modules, and there are many other module resources available online that a programmer can use so that she does not have to "reinvent the wheel," so to speak.

A schematic diagram of the modular construction of a possible Python program is as follows.

```
Python shell ---> First Module.py ---> Second Module.py, etc. --->
Standard Library
>>> commands     <---objects      <---objects      <---objects
results
```

The following example, taken from the code in Example 16.1, shows the composition and use of a module, which happens to be a function, that you create:

If you have not already done so, type the code from Example 16.1 into a file named **arith.py** using your favorite text editor (or rename **math1.py** as **arith.py**). Then type the following lines of Python code at the Python command line:

Example 16.21

```
>>> import arith1
>>> A = arith1.add(3.5,4.2)
>>> B = A + arith1.subtract(16.78,20.45)
>>> C = arith1.multiply(B,5.0)
>>> D = arith1.divide(A,C)
>>> D
0.13487133984028396
>>>
```

Notice what is different about Example 16.1's code and the code in Example 16.19:

1. You use `import` rather than `from…import`.
2. The functions in **arith1** are addressed or touched by referencing `arith1.function _ name`.

EXERCISE 16.11

1. After doing Example 16.21, what result do you get if you type **`b`** on the Python command line, and why?

2. After doing Example 16.21, type **del arith1** on the Python command line. Then type **`D = arith1.divide(A,C)`** and press `<Enter>` on the keyboard. What result do you get, and why?

3. Edit the file **arith1.py**, and comment out all of the return statements in the functions. Then, redo the commands shown in Example 16.21. What is `D` equal to, and why?

A simple example of a library module from the standard library that you can import and use to execute UNIX operating system commands is given next. Type in the commands shown on the Python command line (the output results given in the example may differ from what you see on your screen, depending on the specifics of your system):

Example 16.22a

```
>>> import os
>>> File = os.popen('pwd')
>>> File.read(50)
'/usr/home/bob\n'
>>> for line in os.popen('ls -la f*'): print(line.rstrip())
#Press Enter twice!
...
-rw-r--r-- 1 bob bob 42 Feb 14 23:07 func1.py
-rw-r--r-- 1 bob bob 222 Feb 14 23:08 func1.pyc
-rw-r--r-- 1 bob bob 27 Feb 14 21:54 func2.py
-rw-r--r-- 1 bob bob 207 Feb 14 21:47 func2.pyc
```

16.4 PRACTICAL EXAMPLES

To begin this section, it would be helpful for you to read and try to understand the following two references in the Python online documentation for the release of Python you are using to get a better "top-down" overview of how Python is structured:

1. *Python Language Reference: Release 2.7.X* by Guido van Rossum and Fred L. Drake, Jr.

2. Python standard library, particularly the `sys` and `os` modules.

In the previous sections of this chapter, we provided an overview of the Python language and its syntax via the writing and execution of small (1–25 line) script files and functions. In this section, we will detail some of Python's practical applications in real-world computer programming with larger (25–50 line) user-written modules, functions, and scripts.

As mentioned in Section 16.1, Python can be used to accomplish tasks revolving around shell scripting, systems programming, network and Internet scripting, database programming, systems administration scripting, GUI scripting, scientific and math programming, and data mining.

We will begin by writing script files in Python that accomplish what UNIX shell scripts accomplish, with the goal of familiarizing you with Python.

16.4.1 Another Way of Writing Shell Script Files

If you have not already done so, you should read and do the examples and exercises in Chapters 12 through 15 to review and get a better feel for basic and advanced shell scripting in a UNIX environment. In this section, we do not go over the basics of shell scripting, but provide methods and practical examples (including the rewriting of some of the shell scripts in Chapters 12 through 15) of how to accomplish what UNIX shell scripting accomplishes, but using Python language syntax and structure. The advantages of using Python are that it is a more robust and extensible language, with many more features and capabilities than any of the UNIX scripting languages.

16.4.1.1 Rewriting Bourne and tcsh Scripts

We start with a simple example of a Bourne shell script rewritten in Python. The Bourne shell script to print out whether a certain directory path exists on our file system or not is given first, and then its Python equivalent is shown in Example 16.22b. You should type in and run both of these code samples, and note the output on your system:

Example 16.22b

```
#!/bin/sh
if [ -d "/usr/bin" ] ; then
    echo "/usr/bin is a directory"
else
    echo "/usr/bin is not a directory"
fi
```

```
#!/usr/local/bin/python
import os
if os.path.isdir("/usr/bin"):
    print "/usr/bin is a directory"
else:
    print "/usr/bin is not a directory"
```

For the Python version of the Bourne shell script, we can run the script file using Way 2 (Script mode). The important things to notice about the Python code are that

1. A standard library module named `os` is imported at the top of the script file.
2. `os.path` is a nested module that provides directory and pathname tools in addition to those tools in the `os` standard library module.

EXERCISE 16.12

1. Give the exact syntax you used on your UNIX shell command line to run the Bourne shell script shown in Example 16.22b.

2. Referring to the online documentation, what other `os` module from the standard library can be used to achieve the same thing as the Python code in Example 16.22b?

3. Edit the Python code for Example 16.22b and substitute the path **/usr/bin/yyy** for **/usr/bin**. What output from the program do you get when you run it after making this change?

Another simple example of a Bourne shell script converted to Python is as follows. The Bourne shell script is from Chapter 12.

Example 16.23

Bourne shell code

```
#!/bin/sh
echo "Enter input: \c"
read line
echo "You entered: $line"
echo "Enter another line: \c"
read word1 word2 word3
echo "The first word is: $word1:"
echo "The second word is: $word2:"
echo "The rest of the line is: $word3:"
exit 0
```

Python code

```
#!/usr/local/bin/python
import sys
s = raw_input("Enter input:")
print "You entered:", s
r = raw_input("Enter another line:")
words = r.split(' ')
print "The first word is:", words[0]
print "The second word is:", words[1]
rest = (' '.join(words[2:]))
print "The rest of the line is:", rest
sys.exit() #normal exit status
```

Take note of the Python string methods and slicing used in Example 16.23.

EXERCISE 16.13

1. Give two examples of list indexing or slicing used in Example 16.23.

2. Give two examples of string methods from Example 16.23.

The following is another illustration of taking a Bourne shell script and converting it to Python. The Bourne shell script, `cmdargs _ demo`, is taken from Section 12.4.

Example 16.24

Bourne shell code

```
#!/bin/sh
echo "The command name is: $0."
echo "The number of command line arguments passed as
parameters is: $#."
echo "The value of the command line arguments are: $1 $2 $3 $4
$5 $6 $7 $8 $9."
echo "Another way to display values of all the arguments: $@."
echo "Yet another way is: $*."
exit 0
```

Python code

```
#!/usr/local/bin/python
import sys
x = (sys.argv)
print "The command name is: ", sys.argv[0]
print "The number of command line arguments passed as
parameters is: ", len(sys.argv[1: ])
print "The value of the command line arguments are: ", x[1: ]
print "Another way to display values of all the arguments: ",
sys.argv[1: ]
print "Yet another way is: ", sys.argv[slice(1,9)]
sys.exit ( )
```

Similar to the Bourne shell in syntax and structure, the `C` shell has functional capabilities that can be implemented easily by Python. The following is an example of a script file, `if _ demo1`, taken from Section 14.6.1, whose syntactic structure and program functionality are converted to a Python script file that you should run in Way 2 (Script mode). Notice that, in our Python conversion, we also conditionally check to see if a hidden or dot (.) file has been entered as an argument:

Example 16.25

C shell code

```
#!/bin/csh
if ( ($#argv == 0) | | ($#argv > 1) ) then
    echo "Usage: $0 ordinary_file"
    exit 1
endif
if ( -f $argv[1] ) then
    set filename = $argv[1]
    set fileinfo = 'ls -il $filename'
    set inode = $fileinfo[1]
```

```
        set size = $fileinfo[5]
        echo "Name       Inode      Size"
        echo
        echo "$filename     $inode    $size"
        exit 0
    endif
    echo "$0: argument must be an ordinary file"
    exit 1
```

Python code

```
#!/usr/local/bin/python
import os
import sys
if len(sys.argv) = = 1 or len(sys.
argv) > 2:    #check for no or too many arg(s)
    print "Usage: ", sys.argv[0], "is not an ordinary file"
    sys.exit(1)
if sys.argv[1].startswith('.'):     #added check for dot files
    print "Usage: ", sys.argv[1], "is a dot file"
    sys.exit(1)
if os.path.isfile(sys.
argv[1]):                         #bingo, get stats
    filename = sys.argv[1]
    fileinfo = os.stat(filename)
    print "Filename   inode    size"
    Print " "
    print filename, fileinfo.st_ino, fileinfo.st_size
    sys.exit(0)
else:                             # argument must be a directory
    print sys.argv[1], " argument must be an ordinary file"
    sys.exit(1)
```

16.4.2 Basic User File Maintenance

A very useful and important aspect of your interaction with a computer is how effectively you can maintain the files on your system. The Python standard library, and many user-written libraries and modules, can help you do this efficiently in UNIX. You can utilize the extensive syntax and multiparadigm programming capabilities of Python to go far beyond the capabilities of doing operating system and file maintenance available in any of the UNIX shell programs. This section assumes that you have already read and done the exercises and problems in Chapters 4 through 9 that deal with file manipulation. User file maintenance consists of creating, saving, organizing, and deleting files on your system, in your own account. Chapter 23 describes and details how the same things can be done on a system-wide level by the person that administers your computer. The following sections are a good preparation for what is shown in that chapter.

16.4.2.1 Manually Mounting and Unmounting a USB Thumb Drive with Python

A very common and frequently done basic user file maintenance operation performed by an ordinary user is the deployment of a USB thumb drive on their system. This is usually done to maintain files, to save or archive them, or perhaps to be able to transfer files between one computer and another. A simple example of how to use Python to mount and unmount a USB thumb drive manually on a UNIX computer (in our case the base UNIX system, PC-BSD) follows.

A few preliminary procedures and issues must be considered first before actually executing the Python code in Example 16.26 to achieve the mounting and unmounting of the USB thumb drive to your user account file system. We assume that the file system structure on the USB thumb drive is an MS-DOS file system.

- Many UNIX systems with a GUI interface like Gnome or KDE (for example, Oracle Solaris, the current Solaris-family UNIX system) automatically mount the USB thumb drive on the file system, and make it available as a desktop icon. Therefore, it is not feasible to carry out the following example (Example 16.26) on those types of system. This is similar to what you would experience on non-UNIX systems. PC-BSD does *not* do this for security and other administrative reasons. PC-BSD under KDE does make a system "tray" icon available, known as the "mount tray," to allow you to semiautomatically mount items such as USB thumb drives and so forth. It can be found in the lower-right corner of the screen, and if you right-click on it, you can make several pop-up menu choices affecting the mount tray. Also, from the KDE desktop kickoff applications launcher, you can choose from the "System Settings" menu. One of the "System Settings" icons is "Removable Devices." When you click on this icon, you can turn automounting on or off, and also configure other settings for removable media. By default, automounting is off. This setting is independent of what the mount tray does. If you turn off automounting and quit the mount tray, as shown in Example 16.26, mounting of removable media such as USB thumb drives or hard drives can be done manually.

  The Python code in Example 16.26 will allow you to do the same mounting and unmounting operations that the mount tray does, but manually. With both mount tray mounting and manual mounting, the file system found on the USB thumb drive will be mounted in your own branch of the file system, and you will own the files on the USB thumb drive.

- When you plug a USB thumb drive into one of the USB connectors on your computer, it shows up as a device entry in **/dev**. If it does not, *do not use it.* Most popular brands of USB thumb drive can be used with both our Solaris and PC-BSD systems.

These USB thumb drives should be formatted to FAT32, which is usually the default formatting.

- You must have superuser privilege on the UNIX system on which you are mounting the USB thumb drive.

Example 16.26

0. If you are using PC-BSD, exit the mount tray by right-clicking on its icon at the lower right of the screen display, and making the pop-up menu choice **More Options** -> Close Tray.
   On Solaris, it is not feasible to do a similar operation.
1. Change your current working directory to **/dev**, and list the files in that directory. The USB devices already attached to your computer should have names like **da0**, and so on. Note their name(s).
2. Plug in your USB thumb drive.
3. List the files again in **/dev**. Note the name of the new device added to the file list. On our system, the new device is listed as **da0s1**. This logical device name shows the primary partition on the USB thumb drive as **s1**. If nothing new has been added, get another USB thumb drive, and try Steps 3 and 4 again, until a new device is listed in **/dev**.
4. Return to your home directory, and create a subdirectory with the name **USB**, under your home directory.
5. Turn on superuser mode.
6. Run Python.
7. At the Python command prompt type:

   ```
   >>>import os
   >>>os.popen('mount -t msdosfs /dev/da0s1 /your_home_
      directory/USB')
   ```

   where **`your _ home _ directory`** is the name of the directory under which you created USB in Step 4.
8. Exit Python. Your USB thumb drive has been mounted at **/your_home_directory/USB**, and the files on it are available at that path. In our case it was **/usr/home/bob/USB**.
9. To transfer files back and forth between the thumb drive and your UNIX computer, you should exit superuser. Then, copy files freely between your home account and the thumb drive. Notice that you own the files on the USB thumb drive!
10. To unmount it, turn superuser back on and in a terminal window type **`umount -f /dev/da1s1`**. Do not remove it from the USB connector on your computer until you have unmounted it!
11. We encourage you to use the mount tray as a GUI tool to mount and unmount your removable media. So do not forget to restart the mount tray by clicking on its icon in the PC-BSD control panel, after you are done with the next examples

and exercises . You can also restart it by typing **`pc-mounttray &`** as superuser in a console window.

EXERCISE 16.14

1. What is the exact syntax of the Python command that would unmount the USB thumb drive?
2. What other file system formats are available, what are their names, and how would you change the commands in Example 16.26 to accommodate those different file formats if they were present on your USB thumb drive?
3. After you properly unmount the USB thumb drive, what is in the directory **USB**? After you unmount it and physically pull the thumb drive out of the computer, does it show up as a device in **/dev**?
4. Try mounting other external USB devices to your file system, such as an Android cell phone, a Nexus 7 tablet, an iPad, an iPhone, an mp3 music player, and so on. What results do you attain? Can any of those devices be mounted as external file systems on your UNIX computer?
5. Try mounting an external USB hard drive (usually found in a powered enclosure). We give more instructions on how to do this procedure in Chapter 23.

16.4.2.2 Backing up Your Files

We will not go into the general necessity of backing up your files on your UNIX system as a part of maintaining that computer system, because the reasons for that should seem pretty obvious to all users.

According to UNIX system professionals, there is an easy-to-remember and important set of considerations you must make when backing up the system as an ordinary user, and perhaps even as the system administrator. This set of considerations can be posed in simple question form as "How, What, Why, When, Where, and Who?" Some of the answers to these simple questions can be dovetailed together, and we give a selected list of example answers as follows:

"How" means on a local disk, to Dropbox, to a USB thumb drive manually, to another computer on your home network, automatically by *cron*, to another hard disk manually, totally, incrementally, to RAID, or any variant and combination of these.

"What" means just some of your personal files, all of them, only certain kinds of documents, your entire home directory, the whole disk drive, multiple disk drives, and so on.

"Why" means deciding on the relative importance of "What" you are backing up.

"When" means hourly, once a day, once a week, once a month, every time you save a particular file, and at what time exactly, like 3 a.m.

"Where" means very much the same thing as "How".

"Who" means you personally, automatically by cron, the designated system administrator, Dropbox.com.

To give you a notion of a prudent strategy to deploy in backing up your own user files, the file that contains the words you are reading right now was archived in the following manner:

1. Saved at regular intervals to the hard drive on a local computer
2. Saved periodically to a USB thumb drive mounted on that local computer
3. Saved periodically to another hard drive on another computer attached to the local area network
4. Saved to Dropbox.com

The following examples and procedures in this section assume you have completed Section 16.4.2.1 in preparation for completing the examples presented next. If you are working on a Sun Solaris-family UNIX system, like Solaris, and have *not* done this example because of the constraints mentioned there, you can still do these examples on your automatically mounted USB thumb drive.

We will use the `rsync` command to accomplish our backup strategies in this section. This command is similar to `cp`, except that it is more efficient and faster.

Most importantly, `rsync` "synchronizes" two files or directory structures so that changes in one are reflected in the `rsync` duplicate, either locally between drives, or remotely over a local area network (LAN) or the Internet.

It can copy locally or to/from another host over any remote shell, particularly `ssh`. It has a large number of options that control every aspect of its behavior and permit very flexible specification of the set of files to be copied. The `rsync` command finds files that need to be transferred using a "quick check" algorithm (by default) that looks for files that have changed in size or in last-modified time. We encourage you to consult the `rsync` manual page for more information.

The general forms of the `rsync` command are

Local:

```
rsync [OPTION(S)...] SRC... [DEST]
```

```
  Across a network:
```

```
Pull: rsync [OPTION(S)...] [USER@]HOST:SRC... [DEST]
Push: rsync [OPTION(S)...] SRC... [USER@]HOST:DEST
```

where `OPTION(S)` are the valid options for the rsync command, `SRC` is the source file or directory, and `DEST` is the destination path.

The next five examples will use Python standard library modules and embed Bourne shell commands in a Python "wrapper" (UNIX shell command(s) embedded in Python code), primarily using **os.system**, to

1. Back up a single file on the hard disk to a mounted USB thumb drive
2. Back up a single directory beneath your home directory on the hard disk to a directory on a mounted USB thumb drive
3. Back up a single directory beneath your home directory on the hard disk to another network location on your local area network in Push mode
4. Back up a directory on the hard disk to a mounted USB thumb drive in a rolling, incremental scheme that creates "snapshots" of the source directory anytime the Python script is run
5. Customize a system command to show permissions of files in the current working directory that match a certain pattern

The following simple example shows you how to use Python to back up a single file on your hard disk to the USB thumb drive you mounted and attached to your system's file system in Example 16.26. It assumes you have an ordinary file in the current working directory named **rsynctest**, and that the destination path on the USB thumbdrive is **/usr/home/bob/USBint**.

Example 16.27

```
>>>import os
>>>os.system('rsync -av rsynctest /usr/home/bob/USBint')
sending incremental file list
rsynctest

sent 192 bytes received 35 bytes 454.00 bytes/sec
total size is 92 speedup is 0.41
>>>
```

The following example shows you how to use Python to back up an entire directory on your hard disk to the USB thumb drive you mounted and attached to your system's file system in Example 16.26. It assumes you have a directory under the current working directory named **syncdir**, and that the destination path on the USB thumb drive is **/usr/home/bob/USBint**.

Example 16.28

```
>>>import os
>>>os.system('rsync -av syncdir /usr/home/bob/USBint')
sending incremental file list
syncdir/
syncdir/Chap16.doc
```

```
syncdir/backup1.py
syncdir/ossystem.py

sent 280,176 bytes received 77 bytes 186,835.33 bytes/sec
total size is 279,855 speedup is 1.00
0
>>>
```

The following example shows you how to use Python to back up an entire directory on your hard disk to a remote location on your local area network. It assumes:

1. You have the **ssh** daemon running on both your local and remote host
2. You have a directory under the current working directory named **syncdir2** with some files in it
3. Where a password is asked for, you type in your password on the remote host
4. That the destination path to the remote host is **bob@192.168.0.7:/Users/b/unix3e**

Example 16.29

```
>>>import os
>>>os.system('rsync -av -e ssh syncdir2 bob@192.168.0.7:/
Users/b/unix3e')
The authenticity of host '192.168.0.7 (192.168.0.7)' can't be
established.
RSA key fingerprint is 64:62:9c:46:2a:ef:ba:7d:45:02:40:6e:b5:
7e:f2:f5.
No matching host key fingerprint found in DNS.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.0.7' (RSA) to the list of
known hosts.
Password:
building file list ... done
syncdir2/
syncdir2/Chap16.doc
syncdir2/backup1.py
syncdir2/ossystem.py

sent 280,171 bytes received 96 bytes 16,985.88 bytes/sec
total size is 279,855 speedup is 1.00
0
>>>
```

EXERCISE 16.15

1. Repeat the operations shown in Examples 16.16 through 16.29, substituting file names and directory paths on your computer system and local network for those shown in the examples. When backing up files to a USB thumb drive is finished,

be sure to unmount that drive before removing it from the USB connector on your machine.

The following example shows you how to create a rolling backup scheme of "snapshots" of a directory on your hard disk, and archive the contents of the directory to multiple (five) backup directories on the USB thumb drive you mounted and attached to your system's file system in Example 16.26. It is assumed that the source directory containing some files exists. Every time you execute this Python script, it recycles the oldest (fifth) archived directory, and creates a new full backup with the `rsync` command:

Example 16.30

```
#!/usr/local/bin/python
import os
import shutil
target = "/usr/home/bob/USBint/"
i = 1
while i <= 5:
    temp_path = target + str(i) + "/"
    if not os.path.exists (temp_path):
        try:
            os.makedirs (temp_path)
            print "Created  " + temp_path
        except:
            print " Could not create  " + temp_path
    i = i + 1
print "Deleting the oldest archive"
shutil.rmtree (target + "5")
print "Recycle the backups"
os.rename (target + "4", target + "5")
os.rename (target + "3", target + "4")
os.rename (target + "2", target + "3")
# Unlike some other rolling snapshot schemes that use cp -al,
  we do full
#backup copies across devices, filesystems, etc. with the following-
os.system('cp -a ' + target + "1" + " " + target + "2")
os.system('rsync -av /usr/home/bob/python/' + " " + target +
"1")
```

Here is another example of carrying out simple system administration using Python. It customize a shell command to show permissions set on files in the current working directory that match a certain pattern:

Example 16.31

```
#!/usr/local/bin/python
import stat, sys, os, string, commands
```

```
try:
    #Getting search pattern from user and assigning it to a
    list
    pattern = raw_input("Enter the file pattern to search
    for:\n")
    #defining a 'find' string and assigning results to a
    variable
    commandString = "find " + pattern
    commandOutput = commands.getoutput(commandString)
    findResults = string.split(commandOutput, "\n")
    #output find results, along with permissions
    print "Files:"
    print commandOutput
    print "********************************"
    for file in findResults:
        mode=stat.S_IMODE(os.lstat(file)[stat.ST_MODE])
        print "\nPermissions for file ", file, ":"
        for level in "USR", "GRP", "OTH":
            for perm in "R", "W", "X":
                if mode & getattr(stat,"S_I"+perm+level):
                    print level, " has ", perm, " permission"
                else:
                    print level, " does NOT have ", perm, "
permission"
except:
    print "Error - check your input of file matching pattern"
```

In summary, we used `rsync` to backup a file and directories from the hard drive to a USB thumb drive, to a remote host on the local network, and in a rolling scheme to a USB thumb drive. We have also illustrated a simple UNIX file maintenance command useful for system administration. In Chapter 23 we will explore more robust strategies and examples of how to back up user files and system files, and carry out common system administration chores using shell commands and Python code.

16.4.3 Graphical User Interface with Python and Tkinter Widgets

Up until this point in the chapter, we have interacted with Python in a text-based manner, where we typed commands into the Python shell, or into a text editor, and executed Python to view the output results as text. In this section, we will build a "widget"-based (widget is short for window gadget) GUI with Python, where we still create the Python code as text, but see the resulting output of the Python script in the form of widget graphics.

To accomplish this, we assume the following:

1. That you are using a GUI environment to interact with your UNIX system, such as KDE or Gnome. In our base UNIX system, PC-BSD, we installed the default KDE desktop environment when we installed the system. Thus, we have a GUI capability in Python.

2. That you have Python 2.7.9 on your PC-BSD, as we showed in Section 16.2.
3. That you are using a BSD-family UNIX, such as our base system, PC-BSD.
4. If you are using a Solaris-family UNIX system, such as Oracle Solaris or OpenIndiana, you can use the graphical version of the IPS package manager, as mentioned in Section 16.2, to update your Python installation to Python 2.7.X. Additionally, the Tkinter graphics package installation is already done for you if you upgrade to Python 2.7.X in Solaris using the IPS package manager. When you upgrade to Python 2.7.X in Solaris, to launch version 2.7.X of Python, you need to type **python2.7** on the command line.

16.4.3.1 Obtaining Tkinter and Basic Widget Construction

The standard widget GUI package that works with Python is called Tkinter. For you to do add widget GUI components and functionality to Python, you must be able to import the Tkinter package. You can easily test to see if your default Python system has the Tkinter package available. On the Python command line, type:

```
>>> import Tkinter
```

If you get an error message that the module does not exist, or some similar error message, then you must install the Tkinter package from the FreeBSD ports repository. Use the kickoff application launcher at the bottom left of the PC-BSD window, and under "Applications> System", launch an XTerm window. In that window, type the following command as superuser:

```
% pkg install py27-tkinter
```

The pkg package manager may update itself to its latest release, and then, by typing **y** to assent to the installation of packages, Tkinter is installed in the proper directories. You can then import Tkinter as a module in Python.

Basically, there are two parts to Tkinter widget GUI script construction. The first part is constructing your widgets, using a set of universal constructor tools and the Tkinter widget module set. The second part is building your application program from Python code, that accomplishes what you want to do, and then "hooking" it to the widget constructors of the first part.

Tkinter widgets can be constructed to display and be hooked up to your Python application code, either using OOP, where the widgets are object instances of master classes of Tkinter widgets, or they can be constructed using a functional programming approach. We choose the latter approach because it is easier for beginners, even though in the last examples we present you should be able to see that OOP is an essential, but not mandatory, part of Tkinter GUI programming.

The general form of using a Tkinter widget is as follows.

```
>>> widget = Widget.method (master, option=value,
option=value,...)
```

where

`widget` is the name assigned in your Python script to the particular instance you are creating and using

`Widget` is the master instance of a widget class in the Tkinter module

`method` is an optional procedure performed on the master instance of the widget

`master` is container to which our instance of the widget is attached

`option` is a graphical entity or modifier that describes your particular instance

`value` is one of the characteristics that the option can take on

The following simple example shows a complete Tkinter Python script, and the widget it creates. You should execute these four lines of code using Way 2 (Script mode):

Example 16.32

```
>>> from Tkinter import *
>>> w = Label(None, text = 'First Python GUI')
>>> w.pack( )
>>> w.mainloop( )
```

To close the widget, just click on the "destroy window" button (in Figure 16.1 in the upper right-hand corner) in your style of GUI window in which the widget was created. You may have to expand the window to see all of the window manipulation buttons. The following universal traits of a Tkinter widget script illustrated by Example 16.32 are

1. Tkinter programming is *event driven*, meaning you invoke Tkinter and put it in a *wait state*, where it waits for an event like a mouse button click on the destroy window button, or a keyboard entry, and so forth. The widgets you create and which remain on screen generally only do so while Tkinter is waiting for an event to happen, or until you destroy the window. You can also construct exit handling events in your script to close the widget and its window.
2. Line 1 of Example 16.32 imports all Tkinter modules into the current session.
3. Line 2 of Example 16.32 assigns the widget class `Label`, with its modifying options to your object w.
4. Line 3 of Example 16.32 uses the Tkinter pack geometry manager to consolidate elements into w. There are three geometry managers available in Tkinter: `pack`, `grid`, and `place`. We will use the `pack` and `grid` geometry managers in our examples listed next. The `grid` geometry manager, which treats every window or frame as a table of rows and columns, gives you greater control over where widgets and their components are placed.

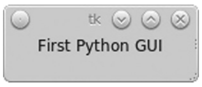


FIGURE 16.1 First Python GUI display.

5. Line 4 of Example 16.32 starts off the event-driven loop that Tkinter enters, and constructs the `Label` widget, with your modifying options, on screen.

16.4.3.2 Tkinter Core Widget Dictionary

The following is a graphical library, or "dictionary," of the core widget classes in Tkinter, along with the Tkinter Python code that generates each core widget class. The graphical library can be used as a visual reference to allow you to choose which kind of widget(s) to which you want to hook your application (see Figures 16.2 through 16.18):

Example 16.33

Label widget:

```
from Tkinter import *
def label():
    master = Tk()
    w = Label(master, text="Label")
    w.pack()
    mainloop()
label()
```

Example 16.34

Button widget:

```
from Tkinter import *
import sys
def quit():
    print "Out of Here"
    sys.exit()
def button():
    master = Tk()
    b = Button(master, text="Button", command=quit)
    b.pack()
    mainloop()
button()
```

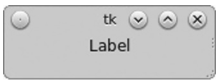


FIGURE 16.2 Tk Label display.

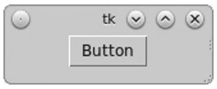


FIGURE 16.3 Tk Button display.

Example 16.35

Canvas widget:

```
from Tkinter import *
def canvas():
    master = Tk()
    master.title("Canvas")
    w = Canvas(master, width=200, height=100)
    w.pack()
    w.create_line(0, 0, 200, 100)
    w.create_line(0, 100, 200, 0, fill="red", dash=(4, 4))
    w.create_rectangle(50, 25, 150, 75, fill="blue")
    mainloop()
canvas()
```

Example 16.36

Checkbutton widget:

```
from Tkinter import *
def checkbutton():
    master = Tk()
    var = IntVar()
    c = Checkbutton(master, text="Checkbutton", variable=var)
    c.pack()
    mainloop()
checkbutton()
```

Example 16.37

Entry widget:

```
from Tkinter import *
def entry():
    master = Tk()
    master.title("Entry")
    e = Entry(master)
```

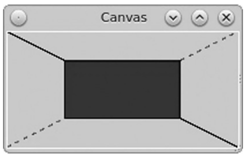


FIGURE 16.4 Tk Canvas display.

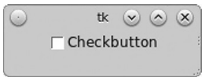


FIGURE 16.5 Tk CheckButton display.

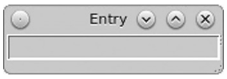


FIGURE 16.6 Tk TextEntry display.

```
    e.pack()
    mainloop()
entry()
```

Example 16.38

Frame widget:

```
from Tkinter import *
def frame():
    master = Tk()
    master.title("Frame")
    Label(text="above Frame separator").pack()
    separator = Frame(height=2, bd=1, relief=SUNKEN)
    separator.pack(fill=X, padx=5, pady=5)
    Label(text="below Frame separator").pack()
    mainloop()
frame()
```

Example 16.39

ListBox widget:

```
from Tkinter import *
def listbox():
    master = Tk()
    master.title("Listbox")
    listbox = Listbox(master)
```

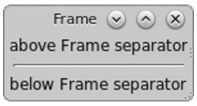


FIGURE 16.7 Tk Frame display.

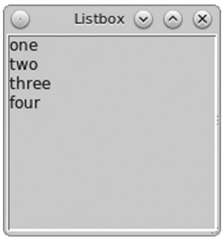


FIGURE 16.8 Tk ListBox display.

```
    listbox.pack()
    for item in ["one", "two", "three", "four"]:
        listbox.insert(END, item)
    mainloop()
listbox()
```

Example 16.40

Menu widget

```
from Tkinter import *
def menu():
    root = Tk()
    root.title("Menu")
    def hello():
        print "I am here"
    # create a toplevel menu
    menubar = Menu(root)
    menubar.add_command(label="1stMenu", command=hello)
    menubar.add_command(label="2ndMenu", command=root.quit)
    # Show them
    root.config(menu=menubar)
    mainloop()
menu()
```

Example 16.41

RadioButton widget

```
from Tkinter import *
def radiobutton():
    master = Tk()
    master.title("RadioButton")
    v = IntVar()
    Radiobutton(master, text="One", variable=v, value=1).
pack(anchor=W)
```

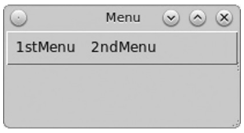


FIGURE 16.9 Tk Menu display.

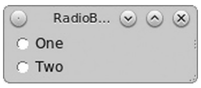


FIGURE 16.10 Tk RadioButton display.

```
    Radiobutton(master, text="Two", variable=v, value=2).
pack(anchor=W)
    mainloop()
radiobutton()
```

Example 16.42

Scale widget

```
from Tkinter import *
def scale():
    master = Tk()
    master.title("Scale")
    w = Scale(master, from_=0, to=100)
    w.pack()
    w = Scale(master, from_=0, to=200, orient=HORIZONTAL)
    w.pack()
    mainloop()
scale()
```

Example 16.43

Scrollbar widget

```
from Tkinter import *
def scrollbar():
    master = Tk()
    master.title("Scrollbar")
```

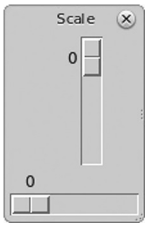
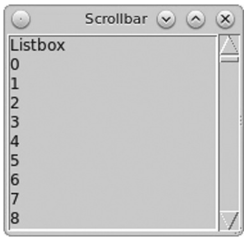


FIGURE 16.11 Tk Scale “Slider” display.



```
Listbox
0
1
2
3
4
5
6
7
8
```

FIGURE 16.12 Tk Scrollbar display.

```
    scrollbar = Scrollbar(master)
    scrollbar.pack(side=RIGHT, fill=Y)
    listbox = Listbox(master, yscrollcommand=scrollbar.set)
    listbox.insert(END, "Listbox")
    for i in range(1000):
        listbox.insert(END, str(i))
    listbox.pack(side=LEFT, fill=BOTH)
    scrollbar.config(command=listbox.yview)
    mainloop()
scrollbar()
```

Example 16.44

PanedWindow widget

```
from Tkinter import *
def panedwindow():
    m1 = PanedWindow()
    m1.pack(fill=BOTH, expand=1)
    left = Label(m1, text="PanedWindow left")
    m1.add(left)
    m2 = PanedWindow(m1, orient=VERTICAL)
    m1.add(m2)
    top = Label(m2, text="PanedWindow top")
    m2.add(top)
    bottom = Label(m2, text="PanedWindow bottom")
    m2.add(bottom)
    mainloop()
panedwindow()
```

Example 16.45

Spinbox widget

```
from Tkinter import *
def spinbox():
```

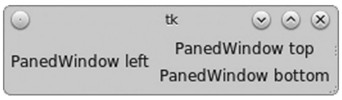


FIGURE 16.13 Tk PanedWindow display.

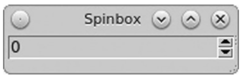


FIGURE 16.14 Tk Spinbox display.

```
    master = Tk()
    master.title("Spinbox")
    w = Spinbox(master, from_=0, to=10)
    w.pack()
    mainloop()
spinbox()
```

Example 16.46

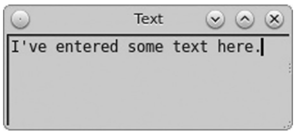
Text widget

```
from Tkinter import *
def text():
    master = Tk()
    master.title("Text")
    textBox = Text(master,wrap=WORD)
    textBox.grid()
    mainloop()
text()
```

Example 16.47

LabelFrame widget

```
from Tkinter import *
def labelframe():
    master = Tk()
    group = LabelFrame(master, text="LabelFrame", padx=5,
    pady=5)
    group.pack(padx=10, pady=10)
    w = Entry(group)
    w.pack()
    mainloop()
labelframe()
```



```
I've entered some text here.
```

FIGURE 16.15 Tk Text Box display.

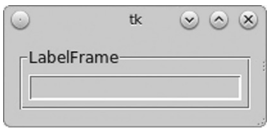


FIGURE 16.16 Tk LabelFrame display.

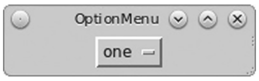


FIGURE 16.17 Tk OptionMenu display.

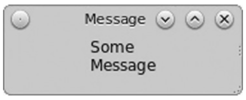


FIGURE 16.18 Tk Message display.

Example 16.48

OptionMenu widget

```
from Tkinter import *
def optionmenu():
    master = Tk()
    master.title("OptionMenu")
    variable = StringVar(master)
    variable.set("one") # default value
    w = OptionMenu(master, variable, "one", "two", "three")
    w.pack()
    mainloop()
optionmenu()
```

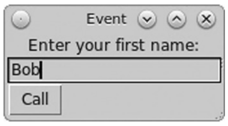
Example 16.49

Message widget

```
from Tkinter import *
def message():
    master = Tk()
    master.title("Message")
    w = Message(master, text="the message")
    w.pack()
    mainloop()
message()
```

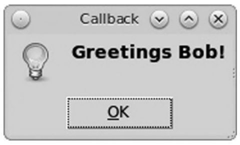
16.4.3.3 Hooking Tkinter Widgets to Applications in Python: Examples

Given the widget library of available core widgets in Tkinter, and armed with your knowledge of Python programming to this point, you are ready to do the following examples. The following simple example allows the user to enter a text string into a dialog box in a widget, and echoes that string as a label in another widget. A line-by-line description/explanation of the code follows the example (see Figures 16.19 and 16.20):



Enter your first name:
Bob

FIGURE 16.19 Tk Event display.



Greetings Bob!

FIGURE 16.20 Tk Callback display.

Example 16.50

Widgets 103

```
from Tkinter import *
from tkMessageBox import showinfo
def reply(name):
    showinfo(title='Callback', message='Greetings %s!' % name)
top = Tk( )
top.title('Event')
Label(top, text="Enter your first name:").pack(side=TOP)
ent = Entry(top)
ent.pack(side=TOP)
btn = Button(top, text="Call", command=(lambda: reply(ent.get(
))))
btn.pack(side=LEFT)
top.mainloop( )
```

A line-by-line description/explanation of Example 16.50:

Line 1. Loads everything from the `Tkinter` widget module.

Line 2. Loads the `tkMessageBox` module, its function `showinfo`, and define `showinfo's` title and message text.

Line 3. Defines the function `reply` that will use `showinfo` to generate the second widget's text.

Line 4. The body of the reply function invoking `showinfo` with its options.

Line 5. Defines the object `top`, and assign the top Tkinter widget to it.

Line 6. Defines the top Tkinter widget title as `Event`.

Line 7. Constructs a `Label` widget, packs it using the `pack` method on `Label`, and positions it.

Line 8. Uses an assignment statement to declare the object `ent`, and constructs it using the `Entry` widget.

Line 9. Packs the `ent` widget using the `pack` method on `Entry`, and positions it.

Line 10. Uses an assignment statement to declare the object `btn`, constructs it using the `Button` widget, and positions and specifies the text to appear in it. Also invokes the reply function defined in Line 3 using the `Lambda` anonymous inline function, which is evaluated when it is called. As the `Lambda` function is invoked, uses the `get` method on `Entry`.

Line 11. Packs and positions the `btn` widget.

Line 12. Starts the event loop.

Even though Example 16.50 uses functional programming syntax and data abstraction, after having examined the line-by-line description/explanation of it, you should begin to see that Tkinter GUI scripts are basically composed of OOP class instance objects. All of the methods in those instances come from the methods on the core widgets in Tkinter.

The following example will construct a Fahrenheit-to-Celsius temperature conversion GUI with Tkinter. A line-by-line description/explanation of the script follows the code (see Figure 16.21):

Example 16.51

Temperature conversion

```
from Tkinter import *
# These colors are set in several places, but this lets us
change in only one.
mainbg = '#8888FF';
activebg = '#AAAAFF';
root = Tk()
root.title('Temp Conversion')
# This grids the widget object where indicated, then returns
it.
def mkgrid(r, c, w):
    w.grid(row=r, column=c, sticky='news')
    return w
# This computes the Celsius temperature from the Fahrenheit.
```

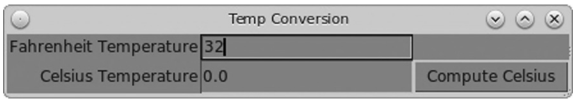


FIGURE 16.21 Tk TempConversion display.

```
def findcel():
    famt = ftmp.get()
    if famt == '':          #not double quote, 2 single quotes
        cent.configure(text='')
    else:
        famt = float(famt)
    camt = (famt - 32) / 1.8
     # A method (configure) applied to an object (cent) that
     is converted to a string (str(camt)).
    cent.configure(text=str(camt))
# The rest hooks the Fahrenheut and Celsius Temperatures into
the grid graphics manager widgets.
flab = mkgrid(0, 0, Label(root, text="Fahrenheit Temperature",
                          anchor='e', bg=mainbg))
clab = mkgrid(1, 0, Label(root, text="Celsius Temperature",
                          anchor='e', bg=mainbg))
ftmp = mkgrid(0, 1, Entry(root, bg=mainbg))
cent = mkgrid(1, 1, Label(root, text="", relief='sunken',
                          anchor='w', bg=mainbg))
elab = mkgrid(0, 2, Label(root, text='', bg=mainbg))
fbut = mkgrid(1, 2, Button(root, text="Compute Celsius",
                         bg=mainbg, activebackground=activebg,
                           command=findcel))
# Starts the root main event loop
root.mainloop()
```

Here is a line-by-line description/explanation of the code of Example 16.51, with comment lines omitted from the count:

Line 1. Imports everything from Tkinter.

Line 2. Specifies the main window background and button color in hex digits.

Line 3. Specifies the active window background in hex digits.

Line 4. Creates the main top-level window, normally referred to as `root`.

Line 5. Applies the title method to `root`, and gives the top-level window the name `Temp Conversion`.

Line 6. Defines the function `mkgrid`, which uses the grid geometry manager to position the widgets.

Line 7. Invokes the grid method on the `w` argument of `mkgrid` with the options of setting row and column numbers, and allocation of extra space within the widget cell.

Line 8. The return of `w` to the main calling script.

Line 9. Hooking widgets to the application begins. Defines a function `findcel` that brings the temperature conversion values into the GUI. The next seven lines of Python calculate the temperature conversion.

Line 10. Uses the `get` method to obtain the input Fahrenheit temperature `ftmp` and assign it to the variable `famt`.

Line 11. If the Fahrenheit variable `famt` is null, then goes to line 12.

Line 12. Uses the `configure` method to set the Celsius variable `cent` to null.

Line 13. Else.

Line 14. Takes the string entered as the Fahrenheit temperature and converts it to a floating point number.

Line 15. The conversion formula for Fahrenheit to Celsius; `camt` is the Celsius temperature.

Line 16. Uses the `configure` method to assign the variable `camt` as a string to the variable `cent`.

Line 17. The next 11 lines of code create the widget instances and hook the calculated temperature conversions to them. Puts a `Label` widget at grid position 0,0, with the text "Fahrenheit Temperature" in it, and assigns that widget to the object `flab`. Uses the function `mkgrid` defined in Line 6 to do this.

Line 18. Continuation of the code on Line 17.

Line 19. Puts a `Label` widget at grid position 1, 0, with the text "Celsius Temperature" in it, and assigns that widget to the object `clab`. Uses the function `mkgrid` defined in Line 6 to do this.

Line 20. Continuation of the code on Line 19.

Line 21. Puts an `Entry` widget at grid position 0, 1, and assigns that widget to object `ftmp`. Uses the function `mkgrid` defined in Line 6 to do this.

Line 22. Puts a `Label` widget at grid position 1,1, allowing the display of the Celsius temperature in it, and assigns that widget to the object `cent`. Uses the function `mkgrid` defined in Line 6 to do this.

Line 23. Continuation of the code on Line 22.

Line 24. Puts a `Label` widget at grid position 0,2, with nothing displayed in it, and assigns that widget to the object `elab`. Uses the function `mkgrid` defined in Line 6 to do this.

Line 25. Puts a `Button` widget at grid position 1,2, with the text "Compute Celsius" in it, and assigns that widget to the object `fbut`. This widget triggers the command to invoke the function `findcel` defined starting on Line 9. Uses the function `mkgrid` defined in Line 6 to do this.

Line 26. Continuation of the code on Line 25.

Line 27. Continuation of the code on Line 26.

Line 28. Starts the main root event loop.

Even though the previous example is a functional program design, you should be able to recognize from this line-by-line description/explanation that the underlying core widgets from Tkinter are OOP classes that we have instanced as objects. The example used methods on those classes, but the structure of our script file was still functional and declarative in nature.

EXERCISE 16.16

1. What is the difference between a Python method and a Python function?
2. In the grid geometry manager, where does the numbering of cells that widgets can be placed in begin, and how do the numbering indices evolve? For instance, in Example 16.51, what does grid position 0,2 mean?
3. What code would put a "quit" button in the grid cell 0,2 instead of a blank label?

16.4.4 Multithreaded Concurrency with Python

The first question you must ask yourself about how and why UNIX functions the way it does is: Given the limited resources and nature of modern computer hardware, *how* does UNIX maximize performance and efficiency for all users of a system? And how can user programs, such as Python, reflect this technique?

The answer for UNIX is

First, by breaking the hardware, that is, CPU(s), main memory, and peripheral memory, into multiple virtual machinery: in short, *virtualization*.

Then, by executing instructions *concurrently* on this virtualized machinery—this means not in any particular sequence, and perhaps even all at once.

Finally, by making sure that the data generated and stored in the file system is *persistent* over time.

Python, as a UNIX tool to create user programs, can achieve concurrency with a mechanism called *threads*, and the concurrent execution of instructions by threads. This is very similar to UNIX system programming with threads.

Similarly to the UNIX system programming concurrency facility, Python threads give you the ability to run several programs concurrently, in a single process. When you create one or more threads in your Python program, they are executed concurrently, independently of each other, and they can share information among them because they are using the same resources of a single process.

These features make Python threads useful in creating Python applications for such things as network programming and the creation of GUI programs.

Python supports threads on PC-BSD and Solaris, and any other systems that uses the POSIX threads library (`pthreads`).

From the Python 2.7.11 documentation at https://docs.python.org/2/c-api/init.html:

> The Python interpreter is not fully thread-safe. In order to support multi-threaded Python programs, there's a global lock, called the global interpreter lock or GIL, that must be held by the current thread before it can safely access Python objects. Without the lock, even the simplest operations could cause problems in a multi-threaded program: for example, when two threads simultaneously increment the reference count of the same object, the reference count could end up being incremented only once instead of twice.
>
> Therefore, the rule exists that only the thread that has acquired the GIL may operate on Python objects or call Python/C API functions. In order to emulate concurrency of execution, the interpreter regularly tries to switch threads. The lock is also released around potentially blocking I/O operations like reading or writing a file, so that other Python threads can run in the meantime.

The major impact this has in terms of Python performance is that Python threads *cannot* take advantage of multiprocessor or multicore processor architectures. Thread switching can only occur between the execution of individual bytecodes in the interpreter. The frequency with which the interpreter checks for thread switching is set by the `sys.setcheckinterval()` function. By default, the interpreter checks for thread switching after every 100 bytecode instructions.

When working with extension modules, the interpreter may invoke functions written in C. Unless specifically written to interact with a threaded Python interpreter, these functions block the execution of all other threads until they complete execution. Thus, a long-running calculation in an extension module may limit the effectiveness of using threads. However, most of the I/O functions in the standard library have been written to work in a threaded environment.

16.4.4.1 *Python Thread Examples Using the Functional Programming Paradigm*

We show both the `thread` (`_thread` in Python 3.X) and `threading` modules, and the examples we present in this section use the `thread` module.

Unless you need the more powerful OOP tools and capabilities in the `threading` module, the choice is largely a matter of programmer and programming team preference, and programming design goals/methodologies. We cover some of the functions offered in the `threading` module in Section 16.4.5.

The basic `thread` module does not impose OOP, and is very easy to use if you are used to functional programming. As mentioned, the `thread` module has been renamed to `_thread` in Python 3.

This module provides low-level primitives for working with multiple threads of control (also called *lightweight processes* or *tasks*) that share their global data space. For synchronization, simple locks (also called *mutexes*) are provided.

16.4.4.1.1 *`thread`* Functions Reference The `thread` module defines the following constants and functions:

`exception thread.error:` Raised on thread-specific errors.

`thread.LockType:` This is the type of lock object.

`thread.start_new_thread(function, args[, kwargs]):` Starts a new thread and return its identifier. The thread executes the function named **`function`** with the argument list **`args`** (which must be a tuple). The optional **`kwargs`** argument specifies a dictionary of keyword arguments. When the function returns, the thread silently exits. When the function terminates with an unhandled exception, a stack trace is printed and then the thread exits (but other threads continue to run).

`thread.interrupt_main():` Raises a `KeyboardInterrupt` exception in the main thread. A subthread can use this function to interrupt the main thread.

`thread.exit():` Raises the `SystemExit` exception. When not caught, this will cause the thread to exit silently.

`thread.allocate_lock():` Returns a new lock object. Methods of locks are described later. The lock is initially unlocked.

`thread.get_ident():` Returns the "thread identifier" of the current thread. This is a nonzero integer. Its value has no direct meaning; it is intended as a magic cookie to be used, for example, to index a dictionary of thread-specific data. Thread identifiers may be recycled when a thread exits and another thread is created.

`thread.stack_size([size]):` Return the thread stack size used when creating new threads.

Lock objects have the following methods:

`lock.acquire([waitflag]):` Without the optional argument, this method acquires the lock unconditionally, if necessary waiting until it is released by another thread (only one thread at a time can acquire a lock—that is the reason for their existence).

`lock.release():` Releases the lock. The lock must have been acquired earlier, but not necessarily by the same thread.

`lock.locked():` Return the status of the lock: `True` if it has been acquired by some thread, `False` if not.

Since the basic `thread` module is a bit simpler than the more advanced `threading` module covered later in the next subsection, we give examples of that first. This module provides a portable interface to whatever threading system is available in your platform: its interfaces work in the same way on PC-BSD and Solaris, and on any system with an installed `pthreads` POSIX threads implementation (including Linux and others). Python scripts that use the Python thread module work on all of these platforms without changing their source code.

16.4.4.1.2 Examples Let us start off by experimenting with a script that deploys the main `thread` interfaces.

The script in Example 16.52 starts successive, one-at-a-time-only threads until you type an **`x`** at the console and then press `<Enter>`:

Example 16.52

```
import thread
def child(tid):
    print 'Started thread', tid
def parent():
    i = 0
    while 1:
        i += 1
        thread.start_new_thread(child, (i,))
        if raw_input() == 'x': break
parent()
% python Example16_52.py
Started thread 1
Started thread 2
Started thread 3
Started thread 4
x
%
```

What exactly is going on in Example 16.52? A single thread is being started, and then it immediately dies, and the program loops indeterminately, allowing you to create successive new threads. Only two thread calls are made in this example: the import of the `thread` module and the call to create the threads. To start a thread, call the `thread.start _ new _ thread` function. This call takes a function (or other callable) object as a tuple argument, and starts a new thread to execute a call to the passed function with the passed arguments.

The following is another example that iterates to create new threads that exist simultaneously, in parallel:

Example 16.53

```
import thread, time
def counter(myId, count):     # function that will run in each
                              thread
    for i in range(count):

        time.sleep(1)         # simulate useful code here
        print '[%s] => %s' % (myId, i)

for i in range(5):            # call start_new_thread 5 times
    thread.start_new_thread(counter, (i, 3)) # loop the newest
                               thread 3 times

time.sleep(5)                 # prevents exit from parent too
                              early

print 'Main thread exiting.'  # all threads are destroyed by
                              default
```

When we execute this script file, we get:

```
% python Example16_53.py
[4] => 0
[2] => 0
[1] => 0
[3] => 0
[0] => 0
[0] => 1
[4] => 1
[1] => 1
[2] => 1
[3] => 1
[2] => 2
[4] => 2
[3] => 2
[0] => 2
[1] => 2
Main thread exiting.
%
```

What exactly is happening in Example 16.53? Five threads are being created and run simultaneously. Within each thread, the counter value from 0 to 4 is being printed at standard output. The `time.sleep(1)` method in the function `counter` is used to simulate code that might be used to do some system programming task(s).

EXERCISE 16.17

If each of the threads you start in Example 16.52 were to perform some operations, after what line in the given code would you put the lines of code that performed that work?

EXERCISE 16.18

In the output of Example 16.53 shown, you will notice that the order in which the value of the `myId` variable that is printed is not always the same. Why is this true?

EXERCISE 16.19

If you run **Example16_53.py** a few times, is the order of the printed value of `myId` the same on each successive run of the program? Why or why not?

16.4.4.2 Python Thread Example Using the OOP Model The `threading` module constructs higher-level threading interfaces on top of the lower-level thread module. If you want your application to make better use of the computational resources of multicore machines, you are advised to use the `multiprocessing` module. However, threading is still an appropriate model if you want to run multiple I/O-bound tasks simultaneously.

16.4.4.2.1 *threading Functions* Reference The `threading` module defines the following functions and objects:

`threading.active_count()`
`threading.activeCount()`

Returns the number of thread objects currently alive. The returned count is equal to the length of the list returned by `enumerate()`.

`threading.Condition()`

A factory function that returns a new condition variable object. A condition variable allows one or more threads to wait until they are notified by another thread.

`threading.current_thread()`
`threading.currentThread()`

Returns the current thread object, corresponding to the caller's thread of control.

`threading.enumerate()`

Returns a list of all thread objects currently alive. The list includes daemonic threads, dummy thread objects created by `current _ thread()`, and the main thread. It excludes terminated threads and threads that have not yet been started.

`threading.Event()`

A factory function that returns a new event object. An event manages a flag that can be set to true with the `set()` method and reset to false with the `clear()` method. The `wait()` method blocks until the flag is true.

`class threading.local`

A class that represents *thread-local data*. These are data whose values are thread specific. To manage thread-local data, just create an instance of `local` (or a subclass) and store attributes on it, for example:

```
mydata = threading.local()
mydata.x = 1
```

The instance's values will be different for separate threads.

`threading.Lock()`

A factory function that returns a new primitive lock object. Once a thread has acquired it, subsequent attempts to acquire it block, until it is released; any thread may release it.

`threading.RLock()`

A factory function that returns a new reentrant lock object.

`threading.Semaphore([value])`

A factory function that returns a new semaphore object.

`threading.BoundedSemaphore([value])`

A factory function that returns a new bounded semaphore object.

`class threading.Thread`

A class that represents a thread of control. This class can be safely subclassed in a limited fashion.

`class threading.Timer`

A thread that executes a function after a specified interval has passed.

`threading.settrace(func)`

Sets a trace function for all threads started from the `threading` module. The function `func` will be passed to `sys.settrace()` for each thread, before its `run()` method is called.

`threading.setprofile(func)`

Sets a profile function for all threads started from the `threading` module. The function `func` will be passed to `sys.setprofile()` for each thread, before its `run()` method is called.

`threading.stack_size([size])`

Returns the thread stack size used when creating new threads.

`exception threading.ThreadError`

Raised for various threading-related errors as described next.

16.4.4.2.2 *`Thread`* Class Objects

The `Thread` class of the `threading` module provides methods applicable to the multithread example presented next, and in general to all Python OOP-based thread synchronization techniques. All of the methods described can be used on the `Thread` class, and are executed *atomically*. For our purposes in this chapter, atomically means without interruption. For illustrations of a wider application of atomic operations, see Chapters 18 through 21.

The `Thread` class represents an activity that is run in a separate thread of control. There are two ways to specify the activity: by passing a callable object to the constructor, or by overriding the `run()` method in a subclass. No other methods (except for the constructor) should be overridden in a subclass. In other words, only override the `__ init __ ()` and `run()` methods of this class.

Once a thread object is created, its activity must be started by calling the thread's `start()` method. This invokes the `run()` method in a separate thread of control.

Once the thread's activity is started, the thread is considered *alive*. It stops being alive when its `run()` method terminates—either normally, or by raising an unhandled exception. The `is _ alive()` method tests whether the thread is alive.

Other threads can call a thread's `join()` method. This blocks the calling thread until the thread whose `join()` method is called is terminated.

A thread has a name. The name can be passed to the constructor, and read or changed through the `name` attribute. A thread can be flagged as a *daemon thread*. The significance of this flag is that the entire Python program exits when only daemon threads are left running. The initial value is inherited from the creating thread. The flag can be set through the `daemon` property.

This constructor should always be called with keyword arguments.

Arguments are:

`group` should be `None`; reserved for future extension when a `ThreadGroup` class is implemented.

`target` is the callable object to be invoked by the `run()` method. Defaults to `None`, meaning nothing is called.

`name` is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

`args` is the argument tuple for the target invocation. Defaults to ().

`kwargs` is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If the subclass overrides the constructor, it must make sure to invoke the base class constructor `(Thread. _ _ init _ _ ())` before doing anything else to the thread.

Methods on the `Thread` class are:

`start()`

Starts the thread's activity. It must be called at most once per thread object. It arranges for the object's `run()` method to be invoked in a separate thread of control. This method will raise a `RuntimeError` if called more than once on the same thread object.

`run()`

Method representing the thread's activity. You may override this method in a subclass. The standard `run()` method invokes the callable object passed to the object's constructor as the `target` argument, if any, with sequential and keyword arguments taken from the `args` and `kwargs` arguments, respectively.

`join([timeout])`

Waits until the thread terminates. This blocks the calling thread until the thread whose `join()` method is called terminates—either normally or through an unhandled exception—or until the optional timeout occurs.

`ident`

The thread identifier of this thread, or `None` if the thread has not been started. This is a nonzero integer.

`is_alive()`
`isAlive()`

Returns whether the thread is alive. This method returns `True` just before the `run()` method starts until just after the **`run()`** method terminates. The module function `enumerate()` returns a list of all alive threads.

`daemon`

A Boolean value indicating whether this thread is a daemon thread (`True`) or not (`False`).

16.4.4.2.3 OOP GUIs and Producer–Consumer Model Threads Threads are extremely important and integral to the Tkinter GUI toolkit, which is illustrated in Section 16.4.3. This also applies in general to GUI libraries such as Qt.

Since many of the functions of a GUI use synchronous I/O, any operation that can block or take a long time to complete must be spawned to run in parallel, so that the central GUI module (the main thread) is always running. Although such children can be run as processes, the efficiency and shared environment model of threads make them ideal for this role. Most GUI toolkits do not allow multiple threads to update the main thread in parallel; updates are best restricted to the main thread.

The two important points to be made about Python threads in a GUI are that the main thread handles all screen graphics updates and that GUI threads must obey the synchronization rules established for general thread concurrency.

All threads in a GUI generally follow what is called the *producer–consumer model*.

This is where one or more objects (the producers) are responsible for placing data into a buffer, and one or more objects (the consumers) are responsible for removing data from that buffer.

The drawbacks to this are as follows.

The producer(s) cannot add more data than the buffer can hold.

The consumers(s) cannot take from an empty buffer,

The actions of all objects *must* be synchronized.

We address more of the issues of the producer–consumer model in Section 16.4.5, and give an example Python program in that section to illustrate the model solution using condition variables.

16.4.4.2.4 *OOP Threads Example*

The following is an example illustrating the basic methodology of using OOP and Python threads. We first describe, in blocks of code, what is happening in the Python code of the example. Then, we present the actual example code in its entirety. Finally, we show sample output when the code is run on the UNIX command line.

It would be very instructive for a beginner to compare what the basic methodology and structure of the following OOP example is compared with the previous Python thread code examples presented.

The components of Example 16.54 are shown as blocks of code as follows (the blocks are indicated as comments on the line of code that begins the block).

Block 1. Import the `Thread` class from the `threading` module

Why do it this way? The `Thread` class, as shown in Section 16.4.4.2.2, contains many useful methods that allow you to construct and manipulate threads. Using it avoids you having to define your own functions or methods to do the same operations. Not doing it this way would mean you would have to write lower-level system programming functions to accomplish thread creation and synchronization, and then, somehow, stitch that code together with higher-level Python functional programming code.

Block 2. Subclass your own thread, named `Threader`, by defining it as a child class based on `Thread`, and also define the constructor properties of it:

Block 3. Define a run method in the `Threader` class. This run method is always executed when we call the start method of any object in our `Threader` class.

The sleep function makes the thread inactive for a definite amount of time. This randomly-timed sleep will ensure that the code will *not* be executed so quickly that we will not be able to notice any changes.

Block 4. The most important block. Create three objects. Call the start method of each object, which, in turn, executes the run method of each object.

You need to call the join method of each object, or the program will terminate before the threads complete their execution.

Example 16.54

```
from threading import Thread                   #Block 1.
from random import randint
import time
class Threader(Thread):                        #Block 2.
    def __init__(self, val):
        ''' Constructor. '''
        Thread.__init__(self)
        self.val = val
    def run(self):                             #Block 3.
        for i in range(1, self.val):
            print('Value %d in thread %s' % (i, self.
            getName()))
            # Sleep for random time
            GoToSleep = randint(1, 5)
            print('%s sleeping fo %d seconds...' % (self.
            getName(), GoToSleep))
            time.sleep(GoToSleep)
if __name__ == '__main__':                     #Block 4.
    # Declare Threader class
    Threader_Object1 = Threader(4)
    Threader_Object1.setName('Thread 1')

    Threader_Object2 = Threader(4)
    Threader_Object2.setName('Thread 2')
    Threader_Object3 = Threader(4)
    Threader_Object3.setName('Thread 3')

    # Run the threads!
    Threader_Object1.start()
    Threader_Object2.start()
    Threader_Object3.start()

# Wait ...
    Threader_Object1.join()
    Threader_Object2.join()
    Threader_Object3.join()

#Exuent ...
    print('Main Terminating...')
```

The output from the program is as follows, when run on the UNIX command line:

```
% python Example16_54.py
Value 1 in thread Thread 1
Thread 1 sleeping for 3 seconds...
Value 1 in thread Thread 3
Thread 3 sleeping for 5 seconds...
Value 1 in thread Thread 2
```

```
Thread 2 sleeping for 2 seconds...
Value 2 in thread Thread 2
Thread 2 sleeping for 2 seconds...
Value 2 in thread Thread 1
Thread 1 sleeping for 1 seconds...
Value 3 in thread Thread 1
Thread 1 sleeping for 4 seconds...
Value 3 in thread Thread 2
Thread 2 sleeping for 4 seconds...
Value 2 in thread Thread 3
Thread 3 sleeping for 5 seconds...
Value 3 in thread Thread 3
Thread 3 sleeping for 3 seconds...
Main Terminating...
%
```

EXERCISE 16.20

If you wanted the threads to do some actual work, in what block of Example 16.54 and exactly where in that block would you put the Python code to accomplish that work?

16.4.5 Talking Threads: The Producer–Consumer Problem Using a Condition Variable

In computing, the producer–consumer problem is a classic example of multiobject synchronization. We addressed some of the issues involved with the producer–consumer model in Section 16.4.4.2.3. In this section, we give more details and a worked example to further illustrate this important computer science concept using Python.

The problem concerns two objects, the producer and the consumer, that share a common, fixed-size buffer used as a queue. The producer produces a piece of data, puts it into the buffer and starts producing again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer) one piece at a time.

The crux of the problem is to make sure that the producer will not add data into the buffer when it is full, and that the consumer will not try to remove data from an empty buffer.

The solution to the problem, for the producer, is to either go to sleep, or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, which starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

In that way, the threads of execution "talk" to each other while they are awake.

16.4.5.1 Signaling Between Threads: Condition Variables

A major component of any threads library, particularly the SUSv3 or POSIX-compliant thread libraries, is the presence of a *condition variable*. Use of a condition variable allows one thread to indicate to other threads that a change in state of a shared variable, or other resource, has happened. For example, if one thread is waiting for another to do something before it can continue.

Condition variables presuppose that some lock is associated with this condition, similar to the features of *mutexes* spoken about in Chapter 21. The lock provides mutual exclusion for accessing the shared variable, but the condition variable indicates changes in the variable's state. This lock must be maintained.

The two principal methods in Python a condition variable has are `wait()` and `notify()`.

16.4.5.2 Python Condition Variable Implementation

To implement a producer–consumer model using a condition variable, we are going to deploy the more advanced `threading` module first introduced in Section 16.4.5.

A description and explanation of the useful functions and methods from that section are repeated here.

16.4.5.2.1 Python Condition Variables and Associated Methods A condition variable has `acquire()` and `release()` methods that call the corresponding methods of the associated lock. It also has a `wait()` method, and `notify()` and `notifyAll()` methods. These three must only be called when the calling thread has acquired the lock, otherwise a `RuntimeError` is raised.

The `wait()` method releases the lock, and then blocks until it is awakened by a `notify()` or `notifyAll()` call for the same condition variable in another thread. Once awakened, it reacquires the lock and returns. It is also possible to specify a timeout.

The `notify()` method wakes up one of the threads waiting for the condition variable, if any are waiting. The `notifyAll()` method wakes up all threads waiting for the condition variable.

The `notify()` and `notifyAll()` methods do not release the lock; this means that the thread or threads awakened will not return from their `wait()` call immediately, but only when the thread that called `notify()` or `notifyAll()` finally relinquishes ownership of the lock.

This technique of programming style, using condition variables, uses the lock to synchronize access to some shared global variable. Threads that are interested in a particular change of state call `wait()` repeatedly until they see the desired state, while threads that modify the state call `notify()` or `notifyAll()` when they change the state in such a way that it could possibly be a desired state for one of the waiters. For example, the following code is a generic producer–consumer situation with unlimited buffer capacity.

```
# Consume one item
cv.acquire()
while not an_item_is_available():
    cv.wait()
get_an_available_item()
cv.release()

# Produce one item
cv.acquire()
make_an_item_available()
```

```
cv.notify()
cv.release()
```

To choose between `notify()` and `notifyAll()`, consider whether one state change may be interesting for only one or several waiting threads. In a typical producer–consumer situation, adding one item to the buffer only needs to wake up one consumer thread.

The following is a listing of methods used to accomplish this.

`threading.Condition()`

A factory function that returns a new condition variable object. A condition variable allows one or more threads to wait until they are notified by another thread.

`acquire(*args)`

Acquires the underlying lock. This method calls the corresponding method on the underlying lock; the return value is whatever that method returns.

`release()`

Releases the underlying lock. This method calls the corresponding method on the underlying lock; there is no return value.

`wait([timeout])`

Waits until notified or until a timeout occurs. If the calling thread has not acquired the lock when this method is called, a `RuntimeError` is raised.

This method releases the underlying lock, and then blocks until it is awakened by a `notify()` or `notifyAll()` call for the same condition variable in another thread, or until the optional timeout occurs. Once awakened or timed out, it reacquires the lock and returns.

When the timeout argument is present and is not `None`, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof).

`notify(n=1)`

By default, wake up one thread waiting on this condition, if any. If the calling thread has not acquired the lock when this method is called, a `RuntimeError` is raised.

This method wakes up at most *n* of the threads waiting for the condition variable; it is a "no operation" (no-op) if no threads are waiting.

An awakened thread does not actually return from its `wait()` call until it can reacquire the lock. Since `notify()` does not release the lock, its caller should.

`notify_all()`
`notifyAll()`

Wakes up all threads waiting on this condition. This method acts similarly to `notify()`, but wakes up all waiting threads instead of one. If the calling thread has not acquired the lock when this method is called, a `RuntimeError` is raised.

16.4.5.3 An Example

The following Python code brings together many of the ideas and syntactic structures from this chapter, to illustrate a Python solution to the producer–consumer problem in a multithreaded program. It uses OOP that deploys the `threading` module and its methods to start two threads, `Producer` and `Consumer`. These threads talk to each other and share the state of the condition variable, which we conveniently name `condition`.

What exactly is happening in Example 16.55? A brief explanation of the layout of the program in major blocks, and a description of the operational heart of the code is as follows:

Block 1: This is the initialization part of the program, where the `threading` module, submodule `Thread`, and class `Condition` are imported. Also, the variables `buffer` and `condition` are initialized. The variable `condition` is derived from the `Condition` class, so that it can use the methods from that class.

Block 2: This defines the class `Producer` derived from the `Thread` module. It also specifies, in a determinate repetition loop, how many times we will add data to the buffer. We are only adding a determinate number of data objects to the buffer in this example. The mechanics of using the condition variable in Producer are explicated here.

Block 3: This defines the class `Consumer` derived from the `Thread` module. How data is removed from the buffer is specified as well. The mechanics of using the condition variable in `Consumer` are explicated here.

Block 4: `Producer` is started, and then `Consumer` is started.

The critical execution order of this program, and how it accomplishes its objectives, are described as follows:

0. The `Producer` thread puts data in the buffer and calls the `notify()` method on the condition instance.
1. The `Consumer` thread checks if buffer is empty before consuming.
2. If the buffer is empty, then call the `wait()` method on condition instance.
3. The `wait()` method blocks the `Consumer` thread and also releases the lock associated with the condition. This lock was held by the `Consumer` thread, so it loses hold of the lock.
4. Unless the `Consumer` thread is notified, it will *not* run.

5. The `Producer` thread can acquire the lock because the lock was released by the `Consumer` thread.
6. The `Producer` thread puts data into the buffer and calls the `notify()` method on the `condition` instance.
7. Once the `notify()` method is done on `condition`, the `Consumer` thread wakes up but does *not* execute.
8. The `notify()` method does not release the lock. Even after `notify()`, the lock is still held by `Producer`.
9. The `Producer` thread explicitly releases the lock by using `condition.release()`.
10. The `Consumer` thread starts running again. Back to Step 0.

Trace these 11 steps through the code we present next:

Example 16.55

```
from threading import Thread, Condition              #Block 1.
import time
import random
buffer = []
MAX_NUM = 10
condition = Condition()
class Producer(Thread):                   #Block 2.
    def run(self):
        global buffer
        for i in range(10):
            condition.acquire()
            if len(buffer) == MAX_NUM:
                print "Queue full, Producer waiting"
                condition.wait()
                print "Queue empty, Producer notified"
            buffer.append(i)
            print "Data Produced", i
            condition.notify()
            condition.release()
            time.sleep(random.random())
class Consumer(Thread):                          #Block 3.
    def run(self):
        global buffer
    for i in range(10):
        condition.acquire()
        while not buffer:
            print "Queue empty, Consumer waiting"
            condition.wait()
            print "Data added to queue, Consumer notified"
        i = buffer.pop(0)
        print "Data Consumed", i
        condition.notify()
        condition.release()
        time.sleep(random.random())
Producer().start()                          #Block 4.
Consumer().start()
```

Output from the above example is as follows:

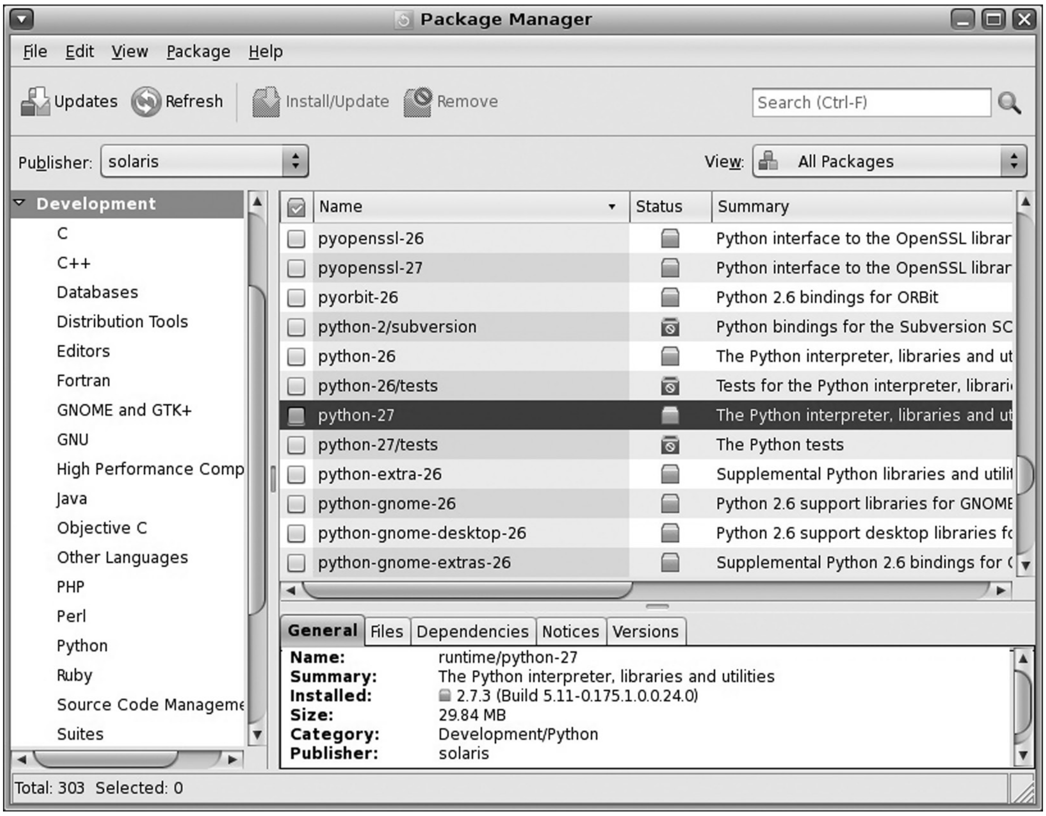
```
% python Example16_55.py
Data Produced 0
Data Consumed 0

Queue empty, Consumer waiting
Data Produced 1
Data added to queue, Consumer notified
Data Consumed 1
Queue empty, Consumer waiting
Data Produced 2
Data added to queue, Consumer notified
Data Consumed 2
Data Produced 3
Data Produced 4
Data Produced 5
Data Consumed 3
Data Produced 6
Data Consumed 4
Data Produced 7
Data Consumed 5
Data Produced 8
Data Produced 9
Data Consumed 6
Data Consumed 7
Data Consumed 8
Data Consumed 9
%
```

EXERCISE 16.21 (refer to Example 16.55)

Briefly state the purpose of the variable `MAX _ NUM`. In the way the program is structured, what value would this variable have to have to be deployed successfully (i.e., prevent a buffer overrun)?

EXERCISE 16.22

As the output of running this program one time shows, the production and consumption of data to and from the buffer is sporadic. That is, sometimes data is produced and



Name	Status	Summary
pyopenssl-26		Python interface to the OpenSSL librar
pyopenssl-27		Python interface to the OpenSSL librar
pyorbit-26		Python 2.6 bindings for ORBit
python-2/subversion		Python bindings for the Subversion SC
python-26		The Python interpreter, libraries and ut
python-26/tests		Tests for the Python interpreter, librari
python-27		The Python interpreter, libraries and ut
python-27/tests		The Python tests
python-extra-26		Supplemental Python libraries and utili
python-gnome-26		Python 2.6 support libraries for GNOME
python-gnome-desktop-26		Python 2.6 support desktop libraries fc
python-gnome-extras-26		Supplemental Python 2.6 bindings for (

Name:	runtime/python-27
Summary:	The Python interpreter, libraries and utilities
Installed:	2.7.3 (Build 5.11-0.175.1.0.0.24.0)
Size:	29.84 MB
Category:	Development/Python
Publisher:	solaris

FIGURE 16.22 Package manager display for Python 2.7.

then immediately consumed, and sometimes data is produced and not immediately consumed. Also, you will notice that if you do several runs of the program, each run may yield different patterns of production and consumption (Figure 16.22). Why is this so?

SUMMARY

We give a broad introduction to the Python programming language, using Python version 2.7.x. We illustrate all of its programming capabilities and syntactic structure, in the context of the three predominant computer programming paradigms. We show the details of doing a fresh install of version 2.7.x or Version 3.x, or an upgrade of those two versions on our two base systems, PC-BSD and Solaris. We show all of Python basic syntax, including numbers and expressions, variables, statements, getting input from the user, functions, OOP in Python, modules, saving and executing Python scripts, string and sequence operations, and error handling. We also give practical examples, such as another way of writing shell script files, rewriting Bash and tcsh scripts, basic user file maintenance, backing up files, remote copying with `rsync`, and graphics using Tkinter.

TABLE 16.6 Python Syntax and Command Summary

Interactive Help in Python Shell

```
help() Invoke interactive help
help(m) Display help for module m
help(f) Display help for function f
dir(m) Display names in module m
```

Module Import

```
import module_name
from module_name import name, ...
from module_name import *
```

Common Data Types

Type	Description	Literal Ex
`int`	32---bit Integer	`3, -4`
`long`	Integer > 32 bits	`101L`
`float`	Floating point number	`3.0, -6.55`
`complex`	Complex number	`1.2J`
`bool`	Boolean	`True, False`
`str`	Character sequence	`'Python'`
`tuple`	Immutable sequence	`(2, 4, 7)`
`list`	Mutable sequence	`[2, x, 3.1]`
`dict`	Mapping	`{ x:2, y5}`

Common Syntax Structures

Assignment Statement

```
var = exp
```

Console Input/Output

```
var = input( [prompt] )
var = raw_input( [prompt] )
print exp[,]
```

Selection

```
if (boolean_exp):
    stmt ...
[elif (boolean_exp):
  stmt ...]
[else:
   stmt ...]
```

Repetition

```
while (boolean_exp):
    stmt …
```

Iteration

```
for var in iterableable_object(sequence suite):
    stmt  …
```

Function Definition

```
def function_name( parmameters ):
    stmt …
```

Function Call

```
function_name( arguments )
```

(*Continued*)

TABLE 16.6 (CONTINUED) Python Syntax and Command Summary

Class Definition

```
class Class_name [ (super_class) ]:
    [ class variables ]
    def method_name( self, parameters ):
        stmt...
```

Object Instantiation

```
obj_ref = Class_name( arguments )
```

Method Invocation

```
obj_ref.method_name( arguments )
```

Exception Handling

```
try:
    stmt ...
except [exception_type] [, var]:
    stmt  ...
```

Common Built-in Functions

Function	Returns
`abs(x)`	Absolute value of *x*
`dict()`	Empty dictionary, e.g.: `d = dict()`
`float(x)`	int or string *x* as float
`id(obj)`	memory addr of *obj*
`int (x)`	float or string *x* as int
`len(s)`	Number of items in sequence *s*
`list()`	Empty list, eg: m = list()
`max(s)`	Maximum value of items in *s*
`min(s)`	Minimum value of items in *s*
`open(f)`	Open filename *f* for input
`ord(c)`	ASCII code of *c*
`pow(x,y)`	x ** y
`range(x)`	A list of x ints 0 to *x* --- 1
`round(x,n)`	float x rounded to n places
`str(obj)`	str representation of *obj*
`sum(s)`	Sum of numeric sequence *s*
`tuple(items)`	tuple of *items*
`type(obj)`	Data type of *obj*

Common Math Module Functions

Function	Returns (all float)
`ceil(x)`	Smallest whole nbr >= *x*
`cos(x)`	Cosine of *x* radians
`degrees(x)`	*x* radians in degrees
`radians(x)`	*x* degrees in radians
`exp(x)`	e ** *x*
`floor(x)`	Largest whole nbr <= *x*

TABLE 16.6 (CONTINUED) Python Syntax and Command Summary

`hypot(x, y)`	sqrt(*x* * *x* + *y* * *y*)
`log(x [, base])`	Log of *x* to *base* or natural log if *base* not given
`pow(x, y)`	`x ** y`
`sin(x)`	Sine of *x* radians
`sqrt(x)`	Positive square root of *x*
`tan(x)`	Tangent of *x* radians
`pi`	Math constant pi to 15 sig figs
`e`	Math constant e to 15 sig figs

Common String Methods

S.method()	**Returns (str unless noted)**
`capitalize`	*S* with first char uppercase
`center(w)`	*S* centered in str *w* chars wide
`count(sub)`	int nbr of non---overlapping occurrences of *sub* in *S*
`find(sub)`	int index of first occurrence of *sub* in *S* or ---1 if not found
`isdigit()`	bool `True` if *S* is all digit chars, `False` otherwise
`islower()`, `isupper()`	bool `True` if *S* is all lower/upper case chars, `False` otherwise
`join(seq)`	All items in *seq* concatenated into a str, delimited by *S*
`lower()`, `upper()`	Lower/upper case copy of *S*
`lstrip()` `rstrip()`	Copy of *S* with leading/trailing whitespace removed, or both
`split([sep])`	List of tokens in *S*, delimited by *sep*; if *sep* not given, delimiter is any whitespace

Formatting Numbers as Strings

Syntax: `format_spec % numeric_exp`

`format_spec`

`width` (optional): align in number of columns specified; negative to left---align, precede with 0 to zero---fill

`precision` (optional): show specified digits of precision for floats; 6 is default

`type` (required): `d` (decimal int), `f` (float), `s` (string), e (float ⍰ exponential notation)

Examples for x = 123, y = 456.789

```
     % x ---> … 123
     % x ---> 000123
     %8.2f % y ---> … 456.79
    "8.2e" %y---> 4.57e+02
    "-8s"%y "Hello" -> Hello …
```

(*Continued*)

TABLE 16.6 (CONTINUED) Python Syntax and Command Summary

Common List Methods

***L*.method()**	**Result/Returns**
`append(`*obj*`)`	Append *obj* to end of *L*
`count(`*obj*`)`	Returns int nbr of occurrences of *obj* in *L*
`index(`*obj*`)`	Returns index of first occurrence of *obj* in *L*; raises `ValueError` if *obj* not in *L*
`pop([`*index*`])`	Returns item at specified *index* or item at end of *L* if *index* not given; raises `IndexError` if *L* is empty or *index* is out of range
`remove(`*obj*`)`	Removes first occurrence of *objfrom L; raises ValueError if obj is not in L*
`reverse()`	Reverses *L* in place
`sort()`	Sorts *L* in place

Common Tuple Methods

***T*.method()**	**Returns**
`count(`*obj*`)`	Returns nbr of occurrences of *obj* in *T*
`index(`*obj*`)`	Returns index of first occurrenceof *obj* in *T*; raises `ValueError` if *obj* is not in *T*

Common Dictionary Methods

D.method()	**Result/Returns**
`clear()`	Remove all items from *D*
`get(`*k* `[,`*val*`])`	Return *D*`[`*k*`]` if *k* in *D*, else *val*
`has_key(`*k*`)`	Return `True` if *k* in *D*, else `False`
`items()`	Return list of key-value pairs in *D*; each list item is two-item tuple
`keys()`	Return list of *D*'s keys
`pop(`*k*`, [`*val*`])`	Remove key *k*, return mapped value or *val* if *k* not in *D*
`values()`	Return list of *D*'s values

Common File Methods

***F*.method()**	**Result/Returns**
`read([`*n*`])`	Return str of next *n* chars from *F*, or up to EOF if n not given
`readline([`*n*`])`	Return str up to next newline, or at most *n* chars if specified
`readlines()`	Return list of all lines in *F*, where each item is a line
`write(`*s*`)`	Write str *s* to *F*
`writelines(`*L*`)`	Write all str in seq *L* to *F*
`close()`	Closes the file

QUESTIONS AND PROBLEMS

1. Type in the following Python code, with the indentation shown, and note what error messages you get. Then, after each error message, type in the proper indentation, until you can execute all eight lines of code.

```
x= 23
if x==27:
print "no go"
    print "why?"
    elif x ==26:
print "still a no go"
    else:
    print "why?"
```

2.
   a. Take the code shown in Example 16.2 and convert it to a function script file that can be executed using Way 3 (Import script mode). Name the function script file **testcase.py**. Save it in the current working directory. The function should allow you to enter different values of x as an input argument each time it is invoked, and print out results on screen.
   b. How is the function brought into and invoked in Python? Test it with several values of x.
3.
   a. Take the code shown in Example 16.3 and convert it to a function script file that can be executed using Way 3 (Import script mode). Name it **nested.py,** and save it in the current working directory. The function should allow you to enter different values of w, y, and z as input arguments each time it is invoked, and print out the results on screen.
   b. How is the function brought into and invoked in Python? Test it with several values of w, y, and z.
4. From the following psuedocode plan,

```
for i ← 1 to length(A)
    j ← i
    while j > 0 and A[j-1] > A[j]
        swap A[j] and A[j-1]
        j ← j - 1
```

   write and test a script file, using Way 2 (Script mode), that:

   a. Allows the user to pass a list individual numbers (real or integer) held in a list A in random order, as arguments to it, and
   b. Uses for and while repetition structure(s) to sort the numbers from list A into ascending order, so their values can print out from low to high, left to right on the screen.

5. Create a function script file in Python that uses Way 3 (Import script mode) to execute it. Name it **factIter.py,** and have it calculate the factorial of an integer number using the `while` indeterminate repetition structure. Test it on the Python command line with several integer values to the script file.
6. Give a definition of list comprehensions, and how the comprehension functional component in Python works on lists, sets, and dictionaries.
7. Similarly to Example 16.10, write a Python function script file that uses Way 3 (Import script mode) to execute. It must deploy the list function, and when invoked at the Python command line, allows you to input integer numbers for the indices of the matrix, and output the resulting random number matrix.
8. Convert the Python function script file you created in Problem 16.7 into a new Python function script file that writes the random number matrix to a file named **matrixout.txt**, similarly to what is done in Example 16.9.
9.
    a. Convert the script file from Example 16.13 into a new Way 3 (Import script mode) function script file that obtains the filename as an input argument to the function.
    b. Rename the data.txt file that Example 16.13's script file worked on to **data2.txt**. Test your new function script file with the filename **data2.txt**.
    c. What are the values of `collection[2]`, `collection[2][1]`, and `collection[1][1]`? How can you obtain these on the Python command line after your new function script file has run?
10. Using Way 2 (Script mode) to execute it, create a Python dictionary that represents a 3×3, two-dimensional matrix of real numbers of your choice. In the same script file, have it print out the elements of the matrix as

```
row 1
row 2
row 3
```

11. After doing Example 16.14, and using the Python standard library reference documentation, find and make a list of some of the basic set operations available (giving the basic syntax for each).
12. Similarly to Examples 16.8 and 16.20, and using Way 3 (Import script mode):
    a. Write a Python function script file that takes a filename as an argument, and
    b. Uses a robust set of `try:…except:…else:` error handling statements in it to test the filename, and
    c. Opens the file named in a., reads a collection of integers from the file into a list, and

d. Converts all of the list elements to integer numbers, and

e. Produces a new integer list of the squares of all the integers, and finally

f. Prints the list of the squares of all the integers.

Then, test the function for your error handling statements with erroneous file names, or names of files that are not in the current working directory.

13. Modify the code in **arith1.py** from Example 16.19 so that the variable `c` is declared as a global variable in all four functions (add, subtract, multiply, and divide). For example:

```
def add(a,b):
    global c
    c = a + b
#NO RETURN STATEMENT!
```

Then:

a. Save the file as **arith2.py**.

b. Import **arith2**, and redo in-chapter Exercise 16.10.

c. Explain the results you get when you type **`>>>arith2.c`** on the Python command line at any particular point after you have imported and invoked the file **arith2** with some numeric arguments.

14. Take the Bourne shell script named **cmdargs_demo**, shown in Section 12.4, and convert it to Python code that runs using Way 2 (Script mode). Your Python conversion should find the byte sizes of all ordinary (regular) and dot (.) files in the directory specified as an argument to the command, and add them up to print a total. It should skip over subdirectories. It should include the branching structure for error handling statements from the original Bourne shell script.

Hint: The following line of Python code adds up or accumulates the running total of file byte sizes of ordinary and dot files in the directory specified:

```
x = sum(os.path.getsize(f) for f in os.listdir(directory) if
os.path.isfile(f))
```

15. Write a Way 2 (Script mode) Python script that takes an ordinary file in the current working directory as an argument, and removes the file from the file system if its size is zero bytes. Otherwise, the script should display the file's name, size, number of hard links, and inode number (in this order) on one line, with a header, similarly to what is shown in Example 16.25. Your script must do appropriate exception handling; for example, it should not work on hidden files or on files which are subdirectories in the current working directory.

16. Following Example 16.30, substitute your own selected directory name on your UNIX system as the source for backup, and back up all the files in that selected directory to

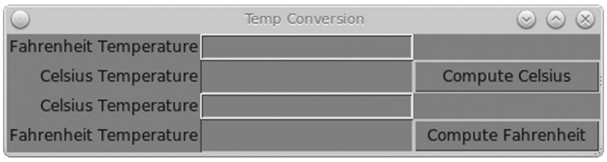


FIGURE 16.23 TempConversion cell layout.

a mounted USB thumb drive on your system in a rolling scheme of three directories. Then, make changes to the source files and directory on the hard drive, and run the script file again to see how the changes have been synchronized in the files and directory on the USB thumb drive.

17. Beginning with the Tkinter script shown in Example 16.51, add Python and Tkinter code to allow the user to do not only Fahrenheit-to-Celsius conversions, but also Celsius-to-Fahrenheit conversions. The layout of the widgets in the Tkinter GUI for this problem can look similar to Figure 16.23.

18. Add a single operating system call to the Example 16.52 code that executes the `ps` command. Every time you press `<Enter>` and start a new successive thread, that thread should execute the `ps` command with output to **stdout**.

19. Add a single operating system call to the Example 16.53 code that executes the `ping -c 5 google.com` command. Every thread will then ping google.com "simultaneously."

20. Write a program using Python threads that pings 10 different hosts simultaneously.

21. Starting with the Python code from Example 16.55, rewrite the program so that an indeterminate number of values are produced by the producer and consumed by the consumer. Termination of the program can be achieved with `<Ctrl-Z>` if necessary.

CHAPTER 17

UNIX Tools for Software Development

Objectives

- To summarize computer programming languages at different levels
- To discuss interpreted and compiled languages and the compilation process
- To briefly describe the software engineering life cycle
- To discuss UNIX program generation tools for C to perform the following tasks: editing, indenting, compiling (of C, C++, and Java programs), handling module-based software, creating libraries, source code management, and revision control
- To describe UNIX tools for static analysis of C programs: verifying code for portability and profiling
- To discuss UNIX tools for dynamic analysis of C programs: debugging, tracing, and monitoring performance
- To cover the commands and primitives

  `ar`, `emacs`, `g++`, `gcc`, `gdb`, `get`, `git` (and all related commands), `grep`, `help`, `indent`, `javac`, `make`, `nm`, `ranlib`, `rlog`, `strip`, `time`

17.1 INTRODUCTION

A typical UNIX system supports several high-level languages, both interpreted and compiled. These languages include C, C++, Pascal, Java, LISP, and FORTRAN. However, most of the application software for the UNIX platform is developed in the C language, the language in which the UNIX operating system is written. Thus, a range of software engineering tools are available for use in developing software in this language. Many of these tools can also be used for developing software in other programming languages, C++ in particular.

The UNIX operating system has a wealth of software engineering tools for program generation and static and dynamic analysis of programs. They include tools for editing source code, indenting source code, compiling and linking, handling module-based software, creating libraries, profiling, verifying source code for portability, source code management, debugging, tracing, and performance monitoring. In this chapter, we describe some of the commonly used tools in the development of C-based software. The extent of discussion of these tools varies from brief to detailed, depending on their usefulness and how often they are used in practice. Before discussing these tools, however, we briefly describe various types of languages that can be used to write computer software. In doing so, we also discuss both interpreted and compiled languages.

17.2 COMPUTER PROGRAMMING LANGUAGES

Computer programs can be written in a wide variety of programming languages. The native language of a computer is known as its *machine language*, the language comprising the *instruction set* of the CPU inside the computer. Recall that the instruction set of a CPU consists of instructions that the CPU understands. These instructions enable the performance of various types of operations on data, such as arithmetic, logic, shift, and I/O operations. Today's CPUs are made of bistate devices (devices that operate in *on* or *off* states), so CPU instructions are in the form of 0s and 1s (0 for the off state and 1 for the on state). The total number of instructions for a CPU and the maximum length (in bytes) of an instruction is CPU dependent. Whereas *reduced instruction set computer* (RISC)-based CPUs have several hundred simple instructions, *complex instruction set computer* (CISC)-based CPUs have a much smaller number of complex instructions. A program written in a CPU's machine language is known as *machine program*, commonly known as *machine code*. The machine language programs are the most efficient because they are written in a CPU's native language. However, they are the most difficult to write because the machine language is very different from any spoken language; the programmer has to write these programs in 1s and 0s, and a change in one bit can cause major problems. Debugging machine language programs is a very challenging and time-consuming task. For these reasons, programs today are rarely written in machine languages.

In *assembly language programming*, machine instructions are written in English-like words, called *mnemonics*. Because programs written in assembly language are closer to the English language, they are relatively easier to write and debug. However, these programs must be translated into the machine language of the CPU used in your computer before you can execute them. This process of translation is carried out by a program called an *assembler*. You have to execute a command to run an assembler, with the file containing an assembly language program as its argument. Although assembly languages are becoming less popular, they are still used to write time-critical programs for controlling real-time systems (e.g., the controllers in drilling machines for oil wells) that have limited amounts of main storage.

In an effort to bring programming languages closer to the English language—and make programming and debugging tasks easier—*high-level languages* (HLLs) were developed. Commonly used high-level languages are Ada, C, C++, Java, JavaScript, Python, BASIC, FORTRAN, LISP, SASL, Pascal, and Prolog. Some of these languages are *interpreted* (e.g., JavaScript, LISP, SASL, and all shell scripts), whereas others are *compiled* (e.g., C, C++, and Java). On the one hand, programs written in an interpreted language are executed one instruction at a time by a program called an *interpreter*, without translating them into the machine code for the CPU used in the computer. On the other hand, programs written in compiled languages must be translated into the machine code for the underlying CPU before they are executed. This translation is carried out by a program called a *compiler*, which generates the assembly version of the high-level language program. The assembly version has to go through further translation before the executable code is generated. The compiled languages run many times faster than the interpreted languages because compiled languages are directly executed by the CPU, whereas the interpreted languages are executed by a piece of software (an interpreter).

However, the Java language is not compiled in the traditional sense. Java programs are translated into a form known as the Java *bytecode*, which is then interpreted by an interpreter.

To simplify the task of writing computer programs even more, languages at a higher level even than the HLLs were developed. They include scripting and visual languages such as UNIX shell programming, Perl, Visual BASIC, and Visual C++. Some of these languages are interpreted; others are compiled. Figure 17.1 shows the proximity of various types of programming languages to the computer hardware, ease of their use, and relative speed at which programs are executed.

As the level of programming languages increases, the task of writing programs becomes easier and programs become more readable. The trade-off is that programs written in HLLs take longer to run. For interpreted programs, the increase in program running time is due to the fact that another program (the interpreter) is running the program. For compiled languages, the compilation process takes longer and the resulting machine code is usually much bigger than it would be if written in assembly language by hand. However, time is saved because the ease of programming in HLLs

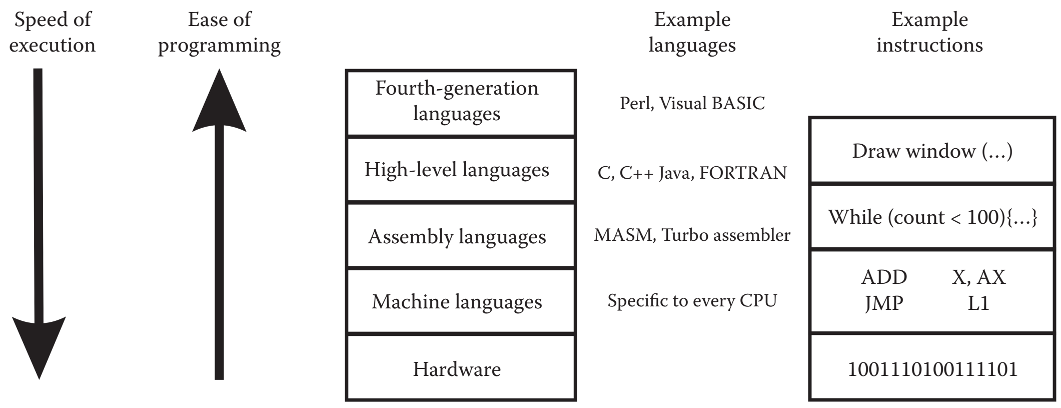


FIGURE 17.1 Levels of programming languages, with examples, ease of programming, and speed of execution.

far outweighs the increase in code size. Figure 17.1 also shows some language statement examples to demonstrate the increased readability of programs as the level of programming languages increases.

17.3 THE COMPILATION PROCESS

Because our focus in this chapter is on UNIX tools—primarily for the C programming language (a compiled language)—we need to describe briefly the compilation process before moving on. As we stated in Section 17.2, computer programs written in compiled languages must be translated to the machine code of the CPU used in the computer system on which they are to execute. This translation is usually a three-step process consisting of *compilation*, *assembly*, and *linking*. The compilation process translates the source code (e.g., a C program) to the corresponding assembly code for the CPU used in the computer system. The assembly code is then translated to the corresponding machine code, known as *object code*. Finally, the object code is translated to the *executable code*. Figure 17.2 outlines the translation process.

The object code consists of machine instructions, but it is not executable because the source program might have used some library functions that the assembler cannot resolve

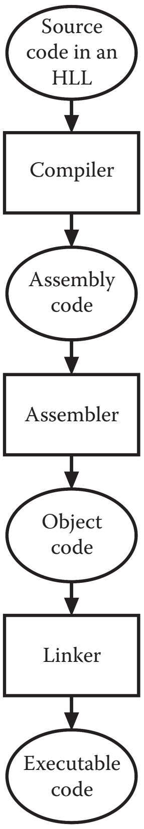


FIGURE 17.2 The process of translating a high-level language program to executable code.

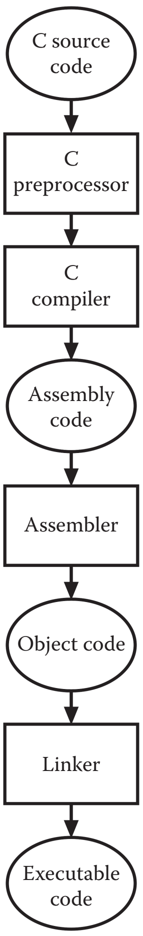


FIGURE 17.3 The process of translating C programs to executable code.

references to, because the code for these functions is not in the source file(s). The linker performs the task of linking (connecting) the object code for a program and the object code in a library, and generates the executable binary code.

The translation of C programs goes through a preprocessing stage before it is compiled. The C preprocessor translates program statements that start with the # sign. Figure 17.3 outlines the compilation process for C programs. The entire translation process is carried out by a single compiler command. We discuss various UNIX compilers later in this chapter.

17.4 THE SOFTWARE ENGINEERING LIFE CYCLE

A software product is developed in a sequence of phases, collectively known as the *software life cycle*. Several life cycle models are available in the literature and used in practice. The life cycle used for a specific product depends on its size, the nature of the software to be developed (scientific, business, etc.), and the design methodology used (object oriented or classical). Some of the commonly used life cycle models are *build-and-fix*, *waterfall*, and *spiral*. The common phases in most life cycle models are *requirement analysis*, *specifications*, *planning*, *design*, *coding*, *testing*, *installation*, and *maintenance*. A full discussion of life cycle models and their phases is outside the scope of this textbook, but we discuss the coding phase in detail—in particular, the UNIX program development tools that can be used in this phase.

The program development process consists of three steps: *program generation*, *static analysis of the source code*, and *dynamic analysis of the executable code*. The purpose of the program generation phase is to create source code and generate the executable code for the source code. Hence, it involves tools for editing text files, indenting the source code properly, compiling the source code, handling module-based software, creating libraries, managing the source code, and controlling revisions. The static analysis phase consists of verifying the source code for portability and measuring metrics related to the source code (e.g., the number of calls to a function and the time taken by each function). The dynamic analysis phase comprises debugging, tracing, and monitoring the performance of the software, including testing it against product requirements. In the rest of this chapter, we describe UNIX tools for all three steps. The depth of discussion on each tool depends on its usefulness for an average software developer, the frequency of its use, and how widely it is available on various UNIX platforms.

17.5 PROGRAM GENERATION TOOLS

The program generation phase consists of creating source code and generating the executable code for it. Hence, it involves tools for editing text files, indenting the source code properly, compiling the source code, handling module-based software, creating libraries, managing the source code, and controlling revisions. We now discuss the UNIX tools for supporting these tasks.

17.5.1 Generating C Source Files

Any text editor can be used to generate C program source files. We discussed the most frequently used UNIX editors, including vim and emacs, in Chapter 3.

17.5.2 Indenting C Source Code

Proper indentation of source code is an important part of good coding practice, primarily because it enhances the readability of the code, and readable code is easier to maintain (debug, correct, and enhance). The best-known indentation style for C programs was proposed by Brian Kernighan and Dennis Ritchie in *The C Programming Language* (1978), the first book on the C language. It is commonly known as the *K&R* (Kernighan and Ritchie) indentation style. Most non-C programmers are not familiar with this style unless they have read the book. The UNIX utility `indent` can be used to indent a C program properly. This utility is available in PC-BSD but not in Solaris. The following is a brief description of the `indent` command.

SYNTAX

`indent input-file [output-file] [options]`

Purpose: This command reads a syntactically correct program specified in **input-file**, indents it according to some commonly accepted C program structure, and saves the formatted program in **output-file** if it is specified in the command line. If the output file is not specified, the formatted version replaces the original version after saving the original version in a file that has the same name as **input-file** and extension **.bak.**

Commonly used options/features:

`/*INDENT OFF*/`	The source code between these two
`/*INDENT ON*/`	comments is not formatted by indent
`-bl`	Format according to Pascal-like syntax
`-br`	Format according to the more commonly used K&R-like syntax, the default setup

Several other options allow you to format your code in various styles. You can specify these options before or after the file names. The `indent` command makes sure that the names of **input-file** and **output-file** are different; if they are the same, it gives an error message and quits. We show a simple use of the command in the following session using the C program file called **second.c**. The `indent` command saves the original contents of the source file **second.c** in the **second.c.bak** file in the current directory.

```
% more second.c
#include <stdio.h>
main()
{
        int i, j;

        for (i=0,j=10; i < j; i++)
        {
                printf("UNIX Rules the Networking World!\n");
        }
}
% indent second.c
% cat second.c
#include <stdio.h>
main()
{
        int             i        , j;

        for (i = 0, j = 10; i < j; i++) {
                printf("UNIX Rules the Networking World!\n");
        }
}
%
```

In the following in-chapter exercise, you will use `indent` to practice indentation of C programs.

EXERCISE 17.1

Create the **second.c** file just described and indent it according to the K&R style by using the `indent` command. What command lines did you use?

17.5.3 Compiling C, C++, and JAVA Programs

Several C compilers are available on UNIX, including cc, xlc, and gcc. The most commonly used C compiler for UNIX is the GNU C/C++ compiler, gcc. This compiler is written for ANSI C, the most recent standard for C language. All C++ compilers, such as the GNU compiler for C++, g++, can also be used to compile C programs. The g++ compiler invokes gcc with options necessary to make it recognize C++ source code. Although we primarily discuss the gcc compiler in this section, we do show a few small examples of the C++ and Java programs and their compilation with the g++ and javac compilers. We use the cc compiler for examples in the rest of the chapter; all of the examples work for the gcc compiler, too. The following is a brief description of the `gcc` command. Most of the options discussed in this section for the gcc compiler also work for the cc compiler. The GNU compilers were preinstalled on PC-BSD, but not available on Solaris. So, as shown in Chapter 23, Section 7.3, we downloaded and installed the GNU compiler package for Solaris with the `pkg` command. Then, on PC-BSD, the default compile command was `gcc48`, and on Solaris it was `gcc`. In the following sessions, we show the compile command simply as `gcc`.

The `gcc` command can be used with or without options. We describe some basic options here and some in later sections of this chapter. One of the commonly used options, even by the beginners, is `-o`. You can use this option to inform gcc that it should store the executable code in a particular file instead of the default **a.out** file. In the following session, we show compilation of the C program in the **first.c** file, with and without the `-o` option. The `gcc first.c` command produces the executable code in the **a.out** file and the `gcc -o slogan first.c` command produces the executable code in the slogan file. The `ls` command is used to show the names of the executable files generated by the two `gcc` commands.

SYNTAX

```
gcc [options] file-list
```

Purpose: This command can be used to invoke the C compilation system. When executed, it preprocesses, compiles, assembles, and links to generate executable code. The executable code is placed in the **a.out** file by default. The command accepts several types of files and processes them according to the options specified in the command line. The files can be archive files (**.a** extension), C source files (**.c** extension), C++ source files (**.C**, **.cc**, or **.cxx** extension), assembler files (**.s** extension), preprocessed files (**.i** extension), or object files (**.o** extension). When a file extension is not recognizable, the command assumes the file to be an object or library/archive file. The files are specified in **file-list**.

Commonly used options/features:

-ansi	Enforce full ANSI conformance.
-c	Suppress the linking phase and keep object files (with the **.o** extension).
-g	Create symbol table, profiling, and debugging information for use with gdb (GNU Debugger).
-llib	Link to the **lib** library.
-mconfig	Optimize code for **config** CPU (**config** can specify a wide variety of CPUs, including Intel 80386, 80486, Motorola 68K series, RS6000, AMD 29K series, and MIPS processors).
-o file	Create executable in **file**, instead of the default file **a.out**.
-O[level]	Optimize. You can specify 0–3 as `level`; generally, the higher the number for `level`, the higher the level of optimization. No optimization is done if `level` is 0.
-pg	Provide profile information to be used with the profiling tool gprof.
-S	Do not assemble or link **.c** files, and leave assembly versions in corresponding files with the **.s** extension.
-v	Verbose mode: Display commands as they are invoked.
-w	Suppress warnings.

```
% cat first.c
#include <stdio.h>

int main ()
{
        printf("UNIX Rules the Networking World!\n");
        return (0);
}
% ls
first.c  second.c
% gcc first.c
% ls
a.out    first.c  second.c
% a.out
UNIX Rules the Networking World!
% gcc -o slogan first.c
% ls
a.out    first.c  second.c slogan
% slogan
UNIX Rules the Networking World!
%
```

If your shell's search path does not include your current directory (.), you will get the message `a.out: Command not found.`, as shown in the following session. If this happens, then you have two options: you can either run the command as `./a.out` (that is, explicitly inform the shell that it should run the **a.out** file in your current directory) or include your current directory in your shell's search path and rerun the command as a.out. The following session illustrates both options. Note that the change in your search path is effective for your current session only; for a permanent change in the search path, you need to change the value of the path variable in your **~/.login** or **~/.cshrc** file (see Chapter 2 for details).

```
% a.out
a.out: Command not found.
% ./a.out
UNIX Rules the Networking World!
% set path = ($path ~)
% a.out
UNIX Rules the Networking World!
%
```

Under Bash, the search path may be changed using the PATH=$PATH:. command. For a permanent change, you need to change the PATH variable's value in the **~/.profile** or **~/.bash_profile** file.

17.5.3.1 Dealing with Multiple Source Files

You can use the gcc command to compile and link multiple C source files and create an executable file, all in a single command line. For example, you can use the following command line to create the executable file called **polish** for the C source files **driver.c**, **stack.c**, and **misc.c**.

```
% gcc driver.c stack.c misc.c -o polish
%
```

If one of the three source files is modified, you need to retype the entire command line, which creates two problems. First, all three files are compiled into their respective object modules, although only one needs recompilation. This results in longer compilation time, particularly if the files are large. Second, retyping the entire line may not be a big problem when you are dealing with three files (as here), but you certainly will not like having to do it with a much larger number of files. To avoid these problems, you should create object modules for all source files and then recompile only those modules that are updated. All the object modules are then linked together to create a single executable file.

You can use the gcc command with the -c option to create object modules for the C source files. When you compile a program with the -c option, the compiler leaves an object file in your current directory. The object file has the name of the source file and an **.o** extension. You can link multiple object files by using another gcc command. In the following session, we compile three source modules—**driver.c**, **stack.c**, and **misc.c**—separately to create their object files, and then use another gcc command to link them and create a single executable file, **polish**.

```
% gcc -c driver.c
% gcc -c stack.c
% gcc -c misc.c
% gcc misc.o stack.o driver.o -o polish
% polish
[output of the program]
%
```

You can also compile multiple files with the `-c` option. In the first of the following command lines, we compile all three source files with a single command to generate the object files. The compiler shows the names of the files as it compiles them. The order in which files are listed in the command line is not important. The second command line links the three object files and generates one executable file, **polish**.

```
% gcc -c driver.c stack.c misc.c
% gcc misc.o stack.o driver.o -o polish
%
```

Now if you update one of the source files, you need to generate only the object file for that source file by using the `gcc  -c` command. Then you link all the object files again using the second of the `gcc` command lines to generate the executable file.

17.5.3.2 Linking Libraries

The C compilers on UNIX systems link appropriate libraries with your program when you compile it. Sometimes, however, you have to tell the compiler explicitly to link the required libraries. You can do so by using the `gcc` command with the `-l` option, immediately followed by the letters in the library name that follow the string `lib` and before the extension. Most libraries are in the **/lib** directory. You have to use a separate `-l` option for each library that you need to link. In the following session, we link the math library (**/lib/libm.a**) to the object code for the program in the **power.c** file. We used the first `gcc` command line to show the error message generated by the compiler if the math library is not linked. The message says that the symbol `pow` is not found in the **power.o** file, the file in which it is used. The name of the math library is **libm.a**, so we use the letter `m`, which follows the string `lib` and precedes the extension, with the `–l` option.

```
% cat power.c
#include <stdio.h>
#include <math.h>

int main()
{
       float  x,y;

       printf ("The program takes x and y from stdin and displays
              x^y.\n");
       printf ("Enter integer x: ");
       scanf  ("%f", &x);
       printf ("Enter integer y: ");
```

```
        scanf  ("%f", &y);
        printf ("x^y is: %6.3f\n", pow((double)x,(double)y));
        return(0);
}
% gcc power.c
/tmp//ccnEwHNH.o: In function 'main':
power_lib.c:(.text+0x73): undefined reference to 'pow'
collect2: ld returned 1 exit status
% gcc power.c -lm -o power
% power
The program takes x and y from stdin and displays x^y.
Enter integer x: 9.82
Enter integer y: 2.3
x^y is: 191.362
%
```

17.5.3.3 Compiling C++ and Java Programs

You can use the gcc compiler to compile C++ programs as well. The file containing C++ source must have one of the following extensions: **.C**, **.CPP**, **.cpp**, **.c++**, or **.cc**.

Java source code is compiled (translated) into Java *bytecode* and is interpreted by the *Java Virtual Machine* (JVM), also known as the *Java Interpreter*. The Java compiler on our UNIX system is called javac, and the JVM is java. Thus, in order to run a Java program in a file, say **Hello.java**, we use the javac compiler to compile it. It produces the Java bytecode and stores it in the **Hello.class** file, which is interpreted with the `java` command, as shown in the following session.

```
% javac Hello.java
$ java Hello
Hello, world!
%
```

In the following in-chapter exercise, you will use the `gcc` and `javac` compiler commands to compile simple C, C++, and Java programs on your UNIX system and run them.

EXERCISE 17.2

Create simple C, C++, and Java programs on your UNIX system. Compile and run them to appreciate the basic working of the two compilers (gcc and javac) and the Java Virtual Machine, java.

17.5.4 Handling Module-Based C Software

Most of the useful C software is divided into multiple source (**.c** and **.h**) files. This software structure has several advantages over a *monolithic* program stored in a single file. First, it leads to more modular software, which results in smaller program files that are less time-consuming to edit, compile, test, and debug. It also allows recompilation of only those source files that are modified, rather than the entire software system. Furthermore, the multimodule structure supports information hiding, the key feature of object-oriented (OO) design and programming.

However, the multimodule implementation also has its disadvantages. First, you must know the files that comprise the entire system, the interdependencies of these files, and the files that have been modified since you created the last executable system. Also, when you are dealing with multimodule C software, compiling multiple files to create an executable sometimes becomes a nuisance because two long command lines have to be typed: one to create object files for all C source files, and the other to link the object files to create one executable file. An easy way out of this inconvenience is to create a simple shell script that does this work. The disadvantage of this technique is that, even if a single source file (or header file) is modified, all object files are recreated, most of them unnecessarily.

UNIX has a much more powerful tool called `make`, which allows you to manage the compilation of multiple modules into an executable. The `make` utility reads a specification file called the *makefile*, that describes how the modules of a software system depend on each other. The `make` utility uses this dependency specification in the makefile and the times when various components were modified, in order to minimize the amount of recompilation. This utility is very useful when your software system consists of tens of files and several executable programs. In such a system, remembering and keeping track of all header, source, object, and executable files can be a nightmare. The following is a brief description of the `make` utility.

SYNTAX

```
make [-f makefile]
```

Purpose: Updates a file based on the dependency relationship stored in a makefile called **makefile**; the dependency relationship is specified in **makefile** in a particular format.

Commonly used options/features:

- `-f` This option allows you to instruct `make` to read interdependency specification from any file; without this option, the file name is treated as **makefile** or **Makefile**.
- `-h` Display a brief description of all options.
- `-n` Do not run any makefile commands; just display them.
- `-s` Run in silent mode, without displaying any messages.

The `make` utility is based on interdependencies of files, target files that need to be built (e.g., executable or object file[s]), and commands that are to be executed to build the target files. These interdependencies, targets, and commands are specified in the makefile as *make rules*. The following is the syntax of a make rule.

SYNTAX

```
target-list: dependency-list
Or
```

```
target-list! dependency-list
Or
target-list:: dependency-list
<Tab> command-list
```

Purpose: The syntax of a make rule

Here, **target-list** is a list of target files separated by one or more spaces, **dependency-list** is a list of files (object, header, source code, etc.) separated by one or more spaces that the target files depend on, and **command-list** is a list of commands—separated by the newline character—that have to be executed to create the target files. Each command in the **command-list** starts with the `<Tab>` character. The comment lines start with the # character. Files in the **target-list** and **dependency-list** may use shell wildcards `?`, `*`, `[]`, and `{}`.

With the `:` operator, a target is considered out of date if its modification time is less than any of the files in the **dependency-list**. If the operator is `!`, the target is always recreated after examining and recreating the sources in the **dependency-list**. With the `::` operator, the targets are always recreated if no sources are specified in **dependency-list**. In case of the `:` and `!` operator, the target is removed if `make` is interrupted, but not in case of the `::` operator. In this book, we use only the most commonly used `:` operator.

The makefile consists of a list of make rules that describe the dependency relationships between files that are used to create an executable file. The `make` utility uses the rules in the makefile to determine which of the files that comprise your program need to be recompiled and relinked to recreate the executable. Thus, for example, if you modify a header (**.h**) file, the `make` utility recompiles all those files that include this header file. The files that contain this header file must be specified in the corresponding makefile.

The following makefile can be used for the power program discussed in Section 17.5.3.

```
% cat makefile
# Sample makefile for the power program
# Remember: each command line starts with a <TAB>
power: power.c
       cc power.c -o power -lm
%
```

If the executable file **power** exists and the source file **power.c** has not been modified since the executable file was created, running `make` will give the message that the executable file is up to date for **power.c**. Therefore, `make` has no need to recompile and relink **power.c**. At times, you will need to force the remaking of an executable because, for example, one of the system header files included in your source has changed. In order to force recreation of the executable, you will need to change the last update time. One commonly used method for doing so is to use the `touch` command, and rerun `make`. The following session illustrates these points.

```
% make
make: 'power' is up to date.
```

```
% touch power.c
% make
cc power.c -o power -lm
%
```

When you use the touch command with one or more existing files as its arguments, it sets their last update time to the current time. When used with a nonexistent file as an argument, it creates a zero-length (i.e., empty) file with that name.

In the following in-chapter exercise, you will use the make command to create an executable for a single source file.

EXERCISE 17.3

Create the executable code for the C program in the **power.c** file and place it in a file called **XpowerY**. Use the make utility to perform this task by using the makefile given previously. Run **XpowerY** to confirm that the program works properly.

In order to show a next-level use of the make utility, we partition the C program in the **power.c** file into two files: **power.c** and **compute.c**. The following session shows the contents of these files. The **compute.c** file contains the compute function, which is called from the main function in **power.c**. To generate the executable in the **power** file, we need to compile the two source files independently and then link them, as shown in the two cc command lines at the end of the session.

```
% cat power.c
#include <stdio.h>

double compute(double x, double y);
int main()
{
        float   x,y;

        printf  ("The program takes x and y from stdin and
                 displays x^y.\n");
        printf  ("Enter integer x: ");
        scanf   ("%f", &x);
        printf  ("Enter integer y: ");
        scanf   ("%f", &y);
        printf  ("x^y is: %6.3f\n", compute(x,y));
}
% cat compute.c
#include <math.h>
double compute (double x, double y)
{
        return (pow ((double) x, (double) y));
}
```

```
% cc -c compute.c power.c
% ls
compute.c compute.o power.c   power.o
% cc compute.o power.o -o power -lm
%
```

The dependency relationship of the two source files is quite simple in this case. To create the executable file **power**, we need two object modules: **power.o** and **compute.o**. If either of the two files **power.c** or **compute.c** is updated, the executable needs to be recreated. Figure 17.4 shows this first cut on the dependency relationship.

The make rule corresponding to this dependency relationship is, therefore, the following. Note that the math library has to be linked because the `compute` function in the **compute.c** file uses the `pow` function in this library.

```
power:      power.o compute.o
            cc power.o compute.o -o power -lm
```

We also know that the object file **power.o** is built from the source file **power.c** and that the object file **compute.o** is built from the source file **compute.c**. Figure 17.5 shows the second cut on the dependency relationship.

Thus, the make rules for creating the two object files are

```
power.o:    power.c
            cc -c power.c
```

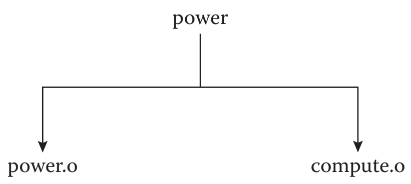


FIGURE 17.4 First cut on the make dependency tree.

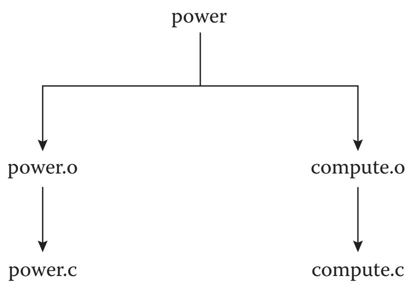


FIGURE 17.5 Second cut on the make dependency tree.

```
compute.o:  compute.c
            cc -c compute.c
```

The final makefile is shown as

```
% cat makefile
power:          power.o compute.o
                cc power.o compute.o -o power -lm
power.o:        power.c
                cc -c power.c
compute.o:      compute.c
                cc -c compute.c
%
```

We then execute the make utility with the preceding makefile:

```
% make
cc power.o compute.o -o power -lm
%
```

In the following in-chapter exercise, you will use the make utility to create the executable code for a C source code that is partitioned into two files.

EXERCISE 17.4

Create the two source files **power.c** and **compute.c** and follow the steps just discussed to create the executable file **power** by using the make utility.

We now change the structure of this software and divide it into six files called **main.c**, **compute.c**, **input.c**, **compute.h**, **input.h**, and **main.h**. The contents of these files are shown in the following session. Note that the **compute.h** and **input.h** files contain declarations (prototypes) of the compute and input functions but not their definitions; the definitions are in the **compute.c** and **input.c** files. The **main.h** file contains two prompts to be displayed to the user. Figure 17.6 shows the new dependency tree.

```
% cat compute.h
/* Declaration/Prototype of the "compute" function */
double compute(double, double);
% cat input.h
/* Declaration/Prototype of the "input" function */
double input (char *);
% cat main.h
/* Declaration of prompts to users */
#define PROMPT1 "Enter the value of x: "
#define PROMPT2 "Enter the value of y: "
```

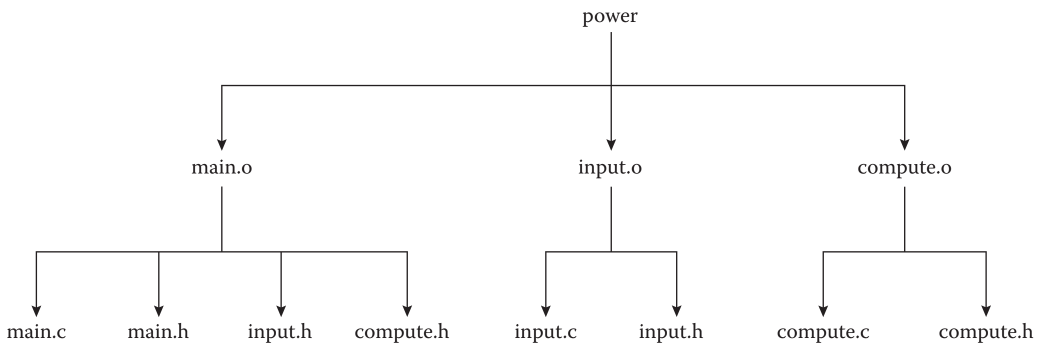


FIGURE 17.6 The make dependency tree for the sample C software.

```
% cat compute.c
#include <math.h>
#include "compute.h"
double compute (double x, double y)
{
        return (pow ((double) x, (double) y));
}
% cat input.c
#include <stdio.h>
#include "input.h"
double input(char *s)
{
        float x;

        printf ("%s", s);
        scanf ("%f", &x);
        return (x);
}
% cat main.c
#include <stdio.h>
#include "main.h"
#include "compute.h"
#include "input.h"

int main()
{
        double x, y;

        printf ("The program takes x and y from stdin and displays
                x^y.\n");
        x = input(PROMPT1);
        y = input(PROMPT2);
        printf ("x^y is: %6.3f\n", compute(x,y));
}
%
```

To generate the executable for the software, you need to generate the object files for the three source files and link them into a single executable. The following commands are needed to accomplish this task. Note that, as before, you need to link the math library while linking the **compute.o** file to generate the executable in the power file.

```
% cc -c main.c input.c compute.c
% cc main.o input.o compute.o -o power -lm
%
```

The makefile corresponding to this dependency relationship is

```
% cat makefile
power:          main.o input.o compute.o
                cc main.o input.o compute.o -o power -lm
main.o:         main.c main.h input.h compute.h
                cc -c main.c
input.o:        input.c input.h
                cc -c input.c
compute.o:      compute.c compute.h
                cc -c compute.c
%
```

The execution of the make command results in the execution of the rules associated with all targets in the makefile.

```
% make
cc main.o input.o compute.o -o power -lm
%
```

In the following in-chapter exercise, you will use the make utility to create an executable for a multimodule C source.

EXERCISE 17.5

Create the three source and header files just discussed, and then use the make command to create the executable in the file **power**. Use the preceding makefile to perform your task.

If the make rules are in a file other than **makefile** (or **Makefile**), you need to run the make command with the -f option, as in make -f my.makefile.

The make rules as shown in the preceding makefile contain some redundant commands that can be removed. The make utility has a predefined rule that invokes the cc -c xxx.c -o xxx.o command for every rule, as in

```
xxx.o: xxx.c zzz.h
       cc -c xxx.c
```

Furthermore, the make utility recognizes that the name of an object file is usually the name of the source file. This capability is known as a *standard dependency*, and because of it you can leave **xxx.c** from the dependency list corresponding to the target **xxx.o**. The following makefile, therefore, works as well as the one given previously.

```
% cat makefile
power:          main.o input.o compute.o
                cc main.o input.o compute.o -o power -lm

main.o:         main.h input.h compute.h

input.o:        input.h

compute.o:      compute.c compute.h
                cc -c compute.c
%
```

Running the make command with this makefile produces the following result:

```
% make
cc main.o input.o compute.o -o power -lm
%
```

The make utility supports simple macros that allow simple text substitution. You must define the macros before using them; they are usually placed at the top of the makefile. A macro definition has the following syntax.

SYNTAX

```
macro_name = text
```

Purpose: Substitute **text** for every occurrence of **$(macro_name)**

With this rule in place, `text` is substituted for every occurrence of `$(macro _ name)` in the rest of the makefile. In addition, the make utility has some built-in macros, such as CFLAGS, that are set to default values and are used by the built-in rules, such as execution of the `cc $(CFLAGS) -c xxx.c -o xxx.o` command for a predefined rule, as previously described.

The default value of the `CFLAGS` macro is usually `-O` (for optimization), but it can be changed to any other flag(s) for the cc compiler. On our system, `CFLAGS` is set to null; that is, there are no default options. The make utility uses several built-in macros for the built-in rules.

The following makefile shows the use of user-defined macros and some useful make rules that can be invoked at the command line. It also shows that the commands for make rules are not always compiler or linker commands; they can be any shell commands.

```
% cat makefile
CC = cc
OPTIONS = -O4 -o
OBJECTS = main.o input.o compute.o
SOURCES = main.c input.c compute.c
HEADERS = main.h input.h compute.h

power:          main.c $(OBJECTS)
                $(CC) $(OPTIONS) power $(OBJECTS) -lm
main.o:         main.c main.h input.h compute.h
input.o:        input.c input.h
compute.o:      compute.c compute.h

all.tar:        $(SOURCES) $(HEADERS) makefile
                tar cvf - $(SOURCES) $(HEADERS) makefile > all.tar

clean:
                rm *.o
%
```

When the `make` command is executed, the commands for the last two targets (`all.tar` and `clean`) are not executed, as these targets do not depend on anything and nothing depends on them. You can invoke the commands associated with these targets by passing the targets as parameters to `make`. The advantage of putting these rules in the makefile is that you do not have to remember which files to archive (by using the `tar` command in this case) and which to remove once the final executable has been created. The `make clean` command invokes the `rm *.o` command to remove all object files that are created in the process of creating the executable for the software. The following session shows the output of `make` when executed with two targets as command line arguments. The `tar` archive is placed in the **all.tar** file.

```
% make all.tar clean
tar cvf - main.c input.c compute.c main.h input.h compute.h
makefile > all.tar
a main.c
a input.c
a compute.c
a main.h
a input.h
a compute.h
a makefile
rm *.o
%
```

In the following in-chapter exercise, you will run the previous sessions on your system to further enhance your understanding of the `make` utility.

EXERCISE 17.6

Use the preceding makefile to create the executable in the file power.

17.5.5 Building Object Files into a Library

The UNIX operating system allows you to archive (bundle) object files into a single file, called a *library*. This process lets you to use the name of one file instead of a number of object files in a makefile and allows function-level software reuse of C programs. The `ar` tool, also called the *librarian*, allows you to perform this task. The following is a brief description of this utility.

SYNTAX

```
ar key archive-name [file-list]
```

Purpose: Allows the creation and manipulation of archives; for example, to create an archive of the object files in **file-list** and store it in the file called **archive-file**.

Commonly used options/features:

`d`	Delete one or more files from an archive
`q`	Append a file to an existing archive
`r`	Create a new archive or overwrite an existing archive
`s`	Force generation of the archive symbol table
`t`	Display the table of contents of an archive
`u`	Update (when used with the **`r`** key) or extract (when used with the **`x`** key) modules only if they are newer than the existing one
`v`	Generate a verbose output
`x`	Extract one or more files from an archive and store them in the current working directory

The **archive-name** must end with the **.a** extension. Once an archive file has been created for a set of object modules, these modules can be accessed by the C compiler and the UNIX loader (ld) by specifying the archive file as an argument. The compiler or the loader automatically links the object modules needed from the archive. The `ld` command can be used to explicitly link object files and libraries.

A *key* is like an option for a command. However, unlike with most UNIX commands, you do not have to insert a hyphen (-) before a key for the `ar` command, but you can use it if you want to. In the following examples of the `ar` command, we do not use a hyphen before a key.

17.5.5.1 Creating an Archive

You can create an archive by using the `ar` command with the `r` key. The following command line creates an archive of the **input.o** and **compute.o** files in **mathlib.a**.

```
% ar r mathlib.a input.o compute.o
%
```

Note that if **mathlib.a** does not exist, it is created. If it already exists and has the **input.o** and **compute.o** modules in it, they are replaced with the ones specified in the command line. Once the archive has been created in your current directory, you can link it to the **main.c** file by using the compiler command such as `cc`, as in:

```
% cc main.c mathlib.a -o power
%
```

You can use the `q` key to append the object modules at the end of an existing archive. Thus, in the following example, the object modules **input.o** and **compute.o** are appended at the end of the existing archive **mathlib.a**. If the **mathlib.a** archive does not exist, it is created.

```
% ar q mathlib.a input.o compute.o
%
```

Once you have created an archive of some object modules, you can remove the original modules in order to save disk space, as in:

```
% rm compute.o input.o
%
```

17.5.5.2 Displaying the Table of Contents

You can display the table of contents of an archive by using the `ar` command with the `t` key. The first of the following commands displays the table of contents of the **mathlib.a** archive and the second displays the archive in verbose format (similar to the output of the `ls –l` command).

```
% ar t mathlib.a
input.o
compute.o
% ar -tv mathlib.a
rw-r--r--    1004/1004       1472 Nov  2 06:37 2014 input.o
rw-r--r--    1004/1004       1208 Nov  2 06:37 2014 compute.o
%
```

17.5.5.3 Deleting Object Modules from an Archive

You can delete one or more object modules from an archive by using the `ar` command with the `d` key. In the following session, the first `ar` command deletes the object module **input.o** from the **mathlib.a** archive, and the second displays the new table of contents confirming the removal of the **input.o** object module from the archive.

```
% ar d mathlib.a input.o
% ar t mathlib.a
compute.o
%
```

Note that creating a brand new archive from scratch is more efficient than modifying an existing archive by using the `d`, `q`, and `r` keys.

17.5.5.4 Extracting Object Modules from an Archive

You can extract one or more object modules from an archive by using the `ar` command with the `x` key. The extracted module remains in the archive. The following command line can be used to extract the object module **cpstr.o** from the **stringlib.a** archive and put it in your current directory.

```
% ar x stringlib.a cpstr.o
%
```

You can run the `ls -l cpstr.o` command to see that the **cpstr.o** object file has been extracted, and the `ar t mathlib.a` command to see that this object file remains a part of the archive.

Although we have shown the use of the `ar` command from the command line, you can also run the command as part of a makefile so that an archive of the object files of a software product is created after the executable file has been created. Doing so allows future use of any general-purpose object modules (one or more functions in these modules) created as part of the software. It is done at the end of a makefile with an explicit make rule, as in:

```
mathlib.a:      input.o compute.o
                ar rv mathlib.a input.o compute.o
                rm input.o compute.o
```

The following makefile is an enhancement of the makefile from the previous section that can be used to create an archive of **input.o** and **compute.o**, called **mathlib.a**. It then removes the **input.o** and **compute.o** files before creating the executable `power` by using the archive **mathlib.a**.

```
% cat makefile
CC = cc
OPTIONS = -O4 -o
OBJECTS = main.o input.o compute.o
SOURCES = main.c input.c compute.c
HEADERS = main.h input.h compute.h

power:          main.o mathlib.a
                $(CC) $(OPTIONS) power main.o mathlib.a -lm
main.o:         main.h input.h compute.h

mathlib.a:      input.o compute.o
                ar rv mathlib.a input.o compute.o
                rm input.o compute.o
```

```
all.tar:        $(SOURCES) $(HEADERS) makefile
                tar cvf - $(SOURCES) $(HEADERS) makefile > all.tar
clean:
                rm *.o
%
```

For each of these rules, the make utility executes a sequence of built-in commands that generate the object module by using the cc command and archives this object module by using the ar command. The following is a sample run of the preceding makefile.

```
% make
cc -O2 -pipe  -c input.c
cc -O2 -pipe  -c compute.c
ar rv mathlib.a input.o compute.o
r - input.o
r - compute.o
rm input.o compute.o
cc -O4 -o power main.o mathlib.a -lm
%
```

In the following in-chapter exercise, you will use the ar command with different options to appreciate its various characteristics in dealing with the libraries of object files.

EXERCISE 17.7

Use the commands just discussed to create an archive, delete an object file from the archive, display the table of contents for an archive, and extract an object file from the archive. Show your work.

17.5.5.5 Working with the MRI Librarian

You can run the ar command with the –M option to invoke the MRI librarian, which allows you to manage libraries with commands from standard input. In the following session, we show how you can use this interface of the ar command to create an archive, add a module to the archive, list modules currently in the archive, delete an archive, extract an archive, save changes in the archive, and close this interface of ar. Note that the default prompt for the MRI librarian is AR >.

```
% ar -M
AR >create math2lib.a
AR >addmod input.o compute.o
AR >list
rw-r--r--    1004/1004       1480 Nov  2 11:15 2014 input.o
rw-r--r--    1004/1004       1232 Nov  2 11:15 2014 compute.o
AR >delete compute.o
AR >list
```

```
rw-r--r--    1004/1004       1480 Nov  2 11:15 2014 input.o
AR >addmod compute.o
AR >list
rw-r--r--    1004/1004       1480 Nov  2 11:15 2014 input.o
rw-r--r--    1004/1004       1232 Nov  2 11:15 2014 compute.o
AR >extract input.o
AR >list
rw-r--r--    1004/1004       1480 Nov  2 11:15 2014 input.o
rw-r--r--    1004/1004       1232 Nov  2 11:15 2014 compute.o
AR >save
AR >end
%
```

In this session, the `create` command is used to create an archive, `addmod` to add one or more object modules to the newly created archive, `list` to list the object files that are currently in the archive, `delete` to delete one or more archives from the archive, `extract` to extract a module from the archive, `save` to save the archive on the disk, and `end` to quite the MRI librarian.

You can work with an existing archive by opening it with the `open` command, as in `open mathlib.a`. You can browse through the manual page of `ar` for more information about the additional features of the MRI librarian. Note that instead of modifying an existing archive (i.e., deleting and adding existing objects from the archive), it is more efficient to remove the existing archive and create a brand new from scratch.

EXERCISE 17.8

Repeat the previous session on your system to understand how the MRI librarian works.

17.5.6 Working with Libraries

A library is an archive of object modules. Working with libraries, therefore, involves creating libraries, ordering modules in a library, and displaying library information. We discussed library creation and manipulation in several ways in the previous section. In this section, we discuss the remaining two operations: ordering archives and displaying library information.

17.5.6.1 Ordering Archives

Object files are not maintained in any particular order in an archive file created by the `ar` command. On some UNIX systems, the caller function must occur before the called function regardless of whether they are in the same or different modules. This condition is a problem because the `cc` and `ld` commands cannot locate object modules unless they are properly ordered. When they cannot locate object modules, these commands display an `undefined symbol` error message when they encounter a call to a function in an object module in an archive. The easiest way to handle this problem is to use the `ranlib` utility, which adds a table of contents to one or more archives that are passed as its parameters.

This utility performs the same task as the ar command with the s key. The following is a brief description of the ranlib utility.

SYNTAX

`ranlib [archive-list]`

Purpose: Adds a table of contents to each archive in **archive-list**

The following ranlib command adds a table of contents to the **mathlib.a** archive. The `ar s mathlib.a` command can also be used to perform the same task.

```
$ ranlib mathlib.a
$
```

17.5.6.2 Displaying Library Information

The nm utility can be used to display the symbol table (names, types, sizes, entry points, etc.) of libraries and object files. The command displays one line for each object (function and global variable) in a library or object file. This output informs you about the functions available in a library and the functions that these library functions depend on. Each output line includes the size (in bytes) of an object, the type of the object (data object, function, file, etc.), scope of the object, and the name of the object. This information is quite useful for debugging libraries. The following is a brief description of the utility.

SYNTAX

`nm key archive-name [file-list]`

Purpose: Allows display of the symbol table of the library and object files specified in **file-list**

Commonly used options/features:

-V Display the version number of the command
-n Display symbols according to their address and not alphabetically
-s Display names of the modules that contain the definitions of the symbols

In the following session, the `nm -V` command is used to display the version of the nm command, and the `nm mathlib.a` command is used to display the information about the **mathlib.a** library that we created in Section 17.4.

```
$ nm -V
GNU nm 2.17.50 [FreeBSD] 2007-07-03
Copyright 2007 Free Software Foundation, Inc.
This program is free software; you may redistribute it under the
terms of the GNU General Public License.  This program has
absolutely no warranty.
```

```
$ nm mathlib.a

input.o:
0000000000000000 T input
                 U printf
                 U scanf

compute.o:
0000000000000000 T compute
                 U pow
%
```

The output of the nm command shows that it contains two object modules: **input.o** and **compute.o**. Further, the `printf`, `scanf`, and `pow` symbols are undefined (U) and the symbols `input` and `compute` are in the text (code) sections of the relevant object modules in the library.

The `nm` and `grep` commands are often run together in order to retrieve information about a specific object. The following commands demonstrate the use of the `nm` command on the math library (**/usr/lib/libm.a**). The command output shows that the `pow` function is in the **e_pow.o** module.

```
% nm -s /usr/lib/libm.a | grep "pow"
imprecise_powl in imprecise.o
powl in imprecise.o
powf in e_powf.o
pow in e_pow.o
0000000000000000 T imprecise_powl
                 U pow
0000000000000000 W powl
e_powf.o:
0000000000000000 T powf
e_pow.o:
0000000000000000 T pow
%
```

17.5.7 Version Control

Studies have shown that about two-thirds of the cost of a software product is spent on maintenance. As we mentioned before, the maintenance of a software product comprises corrective maintenance and enhancement. In corrective maintenance, the errors and bugs found after the deployment of a software product are fixed. In enhancement, the product is enhanced to include more features, such as an improved user interface. Regardless of its type, maintenance means changing and/or revising the source code for the product and generating new executables. This means using a *version control system* (VCS). As you revise source code, you may need to undo changes made to it and go back to an earlier version of the software. Moreover, if an individual or team of programmers is working on a piece of software, they should be able to locally and autonomously maintain editable (modifiable) versions that can be joined together at a convenient time.

Git and GitHub are atomic-level, distributed, content-oriented VCSs. *Atomic level* means that when you take a snapshot of the software package, everything in it is captured in the snapshot at a single instance in time. *Distributed* means that the entire software package you are working with is available to all collaborators locally at all times. *Content oriented* means that when you join different branches of work on the software, only the content of lines in the files along the branches you are merging are considered. Here, *content* means "with form," and *context* means "with meaning." Git only considers content when it operates on your software files and directories; the individual(s), collaborator(s), or integrator(s) of the software project are responsible for the merged context of the files in the software package. The combining of different lines of development of a project that causes content conflicts are indicated by Git with several useful strategies and mechanisms, and their resolution is aided by many tools that are available as add-ons to Git. But only the people writing and testing how the software works, and those people managing that process, are responsible for resolving merged-context conflicts.

EXERCISE 17.9

Give a detailed example of what you think is *content* tracking in a revision control system. Then contrast that example with what you think is *context* tracking.

Git, as a source code maintenance tool, is a software database for tracking changes made to a set of source code files over time. Although programmers most often use it to coordinate changes to software source code, you can use Git to track any kind of content. Git can

- Examine the state of your source code project at earlier points in time
- Show the differences among various states of the project, and the files present at those states
- Split the project development into multiple independent lines, called *branches*, which can evolve separately
- Regularly recombine branches by *merging*, or reconciling, the changes made in two or more branches
- Allow many people to work on a project simultaneously, sharing and combining their work as needed

Git is a member of the newer generation of distributed VCSs. Older systems such as SCCS, RCS, CVS, and Subversion are centralized, meaning that there is a single, central copy of the project content and history to which all users must refer. If the central copy is unavailable, all users must wait until the central copy is online again. Distributed systems such as Git have no central copy. Each user has a complete, independent copy of the entire project history, called a *repository*, and full access to all version control facilities. Network access is only needed to share changes among members of the same development team or group.

Git's distributed nature accommodates many different styles of interaction, or *workflows*. Individuals can share work directly between their personal repositories. Git is the technology behind the *social coding* website GitHub, which includes many well-known open-source projects, most notably for the Linux kernel.

We discuss Git and GitHub in an examples-based tutorial presentation. For Git and GitHub, the following subsections will include:

1. How Git is used, and how it works on a UNIX system
2. A high-level overview of the Git terminology, data structures, objects, and actions
3. Illustrations, both graphical and verbal, of the Git staging model, what a directed acyclic graph (DAG) is, and a finer-grained view of the object store contents
4. A short and a long example of how to create, edit, branch, and merge branches of a Git repository
5. An exploration of GitHub as a remote repository
6. Three basic examples of how to use `git clone`, `git push`, and `git pull` to transfer repository contents between a local repository and GitHub

To get the most useful information out of this section, you are encouraged to first read through the high-level background materials in first three subsections. Then do the examples in the subsequent three sections, as many times as necessary to become comfortable using Git and GitHub. Finally, be sure to do all in-chapter exercises and the problem set on Git and GitHub at the end of this chapter.

17.5.7.1 What Is Git Used for and How Does It Work?

Git is used to manage one or more source code project repositories, each packaged into its own directory that you create for the source code files in that project. A repository is a database containing all the information needed to archive and manage the revisions and history of a project. In Git, as with most VCSs, a repository archives a complete copy of the entire project, with all of the revisions to it.

Git maintains a set of configuration settings and files within each repository. Unlike file data and other repository metadata, configuration settings are not propagated from one repository to another during a *cloning*, or *duplicating*, operation. Instead, Git manages and inspects the configuration and setup information on a per-site, per-user, and per-repository basis. Within a repository, Git maintains two primary data structures, the object store and the index. The object store is designed to be efficiently copied during a clone operation as part of the mechanism that supports a fully distributed VCS. The index is transitory information, is private to a repository, and can be created or modified on demand as needed. All of this repository data is stored at the root of your working directory in a hidden subdirectory named **.git**.

Simply put, with Git and GitHub, you can manage your source code repositories and work in an independent, collaborative, or integrative management way. A majority of the introductory material we present here is aimed at the independent developer. From a learning point of view, you need to first know how to use Git in an independent way. Then we also show some collaborative workflows, particularly with GitHub. We do not touch on the integrative management techniques and commands of Git or GitHub.

EXERCISE 17.10

What do you think the role of an integrator of a project would be, in terms of what a revision control system accomplishes for a software development and maintenance program?

17.5.7.2 Basic Git Terminology

Following is a glossary of basic Git terminology used throughout our examples. This glossary is partitioned into categories that reflect the basic structure of the Git repository itself.

17.5.7.2.1 Top-Level Terminology

Repository: The repository is the *working directory*, and contains inside of itself the source code files you want to maintain and control, the *object store*, and the *index*, as shown in Figure 17.7. The advantage of having the repository self-sufficient inside of its own container is that the container can then be shared locally and globally.

A repository can also be thought of as a collection of *commits*, each of which is an archive of what the project's *working tree* looked like at a past date, whether on your machine or someone else's. It also defines *HEAD*, which identifies the branch or commit the current working tree stemmed from. It contains a set of branches and tags, to identify certain commits by name.

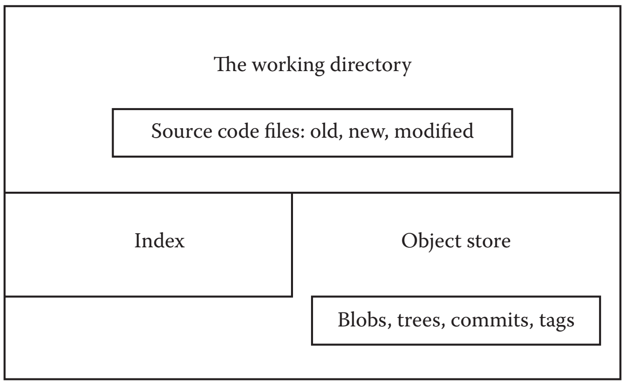

The working directory
Source code files: old, new, modified
Index
Object store
Blobs, trees, commits, tags

FIGURE 17.7 The structure of the repository.

Working directory: The working directory is any directory on your file system that has a repository associated with it, typically indicated by the presence of a subdirectory within it named **.git**. It includes all the files and subdirectories in that directory.

17.5.7.2.2 Primary Data Structures

Object store: Holds the changes in your source code over time, as you perform more commit operations. It is found in the **.git** subdirectory of your working directory. Its primary components or data structures are *blobs*, *trees*, *commits*, and *tags*.

The index (staging area): This is a *cache*, or intermediate area, between your working tree and your repository. You can add changes to the index and build your next commit step by step. When your index content is complete, you then commit from the index. It is also used to keep information during failed merges (your side, their side, and current state). Unlike other, similar tools you may have used, Git does not commit changes directly from the working tree into the repository. Instead, changes are first made in the index. Think of it as a way of double-checking your additions or modifications, one by one, before doing a commit. You can also call it the *staging area*.

17.5.7.2.3 Basic Object Types in the Object Store

Blobs: Each version of a file is represented as a blob. Blob, a contraction of "binary large object," is a term that's commonly used in computing to refer to some variable or file that can contain any data and whose internal structure is ignored by the program. A blob is treated as being opaque. A blob holds a file's data but does not contain any metadata about the file or even its name.

Trees: A tree object represents one level of directory information. It records blob identifiers, pathnames, and a bit of metadata for all the files in one directory. It can also reference other subtree objects recursively and thus build a complete hierarchy of files and subdirectories.

Commits: A commit object holds metadata for each change introduced into the repository, including the author, committer, commit date, and log message. Each commit points to a tree object that captures, in one complete snapshot, the state of the repository at the time the commit was performed. The initial commit, or root commit, has no parent. Most commits have one commit parent. However, as we explain later, a commit, called a merge commit, can reference more than one parent. A commit is the state of your project, or of your working tree at some point in time. The state of HEAD at the time your commit is made becomes that commit's parent. This is what creates the *revision history*.

Tags: A tag is also a name for a commit, similar to a branch, except that it always names the same commit, and can have its own shorthand descriptive text name. A tag object assigns an arbitrary, human-readable name to a specific object, usually a commit. Although the 40-digit-long hexadecimal number is an exact reference to a commit, a more tractable, understandable, and familiar tag name like **Ver-1.0-Beta** is more useful for humans.

17.5.7.2.4 Components and Actions

Working tree: A working tree is a data structure component that represents the state of your source code files and directories at any given point in the history of the repository. Sometimes referenced as the contents of the index, it can be best thought of for beginners in Git as the data structure tree that loads or fills the working directory with its files and directories when you either create or checkout a commit.

Adding: Putting files from the working directory into the index for staging.

Branch: A branch is just a name for a line of commits, also called a reference. It is the parentage of a commit that defines its history—hence, the typical notion of a "branch of project development." It can be simply thought of as a different line of development in the project. A branch in Git is just a "label" that points to a commit. You can get the full history through the parent pointers. A branch by default is only local to your repository.

Checking out: Bringing a branch of the repository into the working directory is called *checking out*.

Directed acyclic graph (DAG): A DAG is a graph of the state of a repository, showing all commits and the parent–child relationships of the commits. It is also a good graphic representation of the branches, tags, and location of HEAD, if those are included in the graph. See Section 17.5.7.4 for more complete and descriptive information.

Master: The main line of development in most repositories is done on a branch called the *master*. It is the default name for the main branch of development.

HEAD: Your repository uses HEAD to define what is currently checked out. If you checkout a branch, HEAD symbolically refers to that branch, indicating that the branch name should be updated after the next commit operation. If you checkout a specific commit, HEAD refers to that commit only. This is referred to as a *detached HEAD*, and occurs, for example, if you check out a tag name.

Clone: A *clone* is a replicated copy of the entire repository, with all of its data structures, files, configurations, and so on.

Merge: The opposite of branch—that is, the fusion of branches and their commits.

EXERCISE 17.11

What do you think would be the quickest and easiest way to delete a local repository on your UNIX system?

17.5.7.3 The Git Staging Model

Git has three main states that your files can be in: *modified*, *staged*, and *committed*. Modified means that you have changed the file but have not committed it to your database yet. Staged means that you have designated a modified file in its current version to go into your next commit snapshot. Committed means that the data is safely stored in your Git repository database. It is held as a data structure consisting of the four types of objects in your object store.

The three main sections of a Git project are seen in Figure 17.8. They are the working directory (where you initially add, create, or modify files), the index or staging area (where you prepare files to be put into the repository), and the repository (that is, in a database held in the **.git** subdirectory of your working directory).

The object store, in your **.git** subdirectory of your working directory, is where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you clone a repository to collaborate with other members of a software development team or group. The working directory contains a single checked-out copy of one version of the project. These files are pulled out of the compressed database in the repository directory and placed on disk for you to use or modify using the `git checkout` command. The index is a file contained in the **.git** subdirectory of your working directory that stores information about what will go into your next commit.

The basic Git workflow is as follows:

1. You place new files into, delete files from, or modify files in your working directory.
2. You stage the files, adding snapshots of them to your index.
3. You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your object store.

If a particular version of a file is in the object store, it is considered committed. If it is modified but has been added to the index, it is staged. And if it was changed since it was checked out but has not been staged, it is modified.

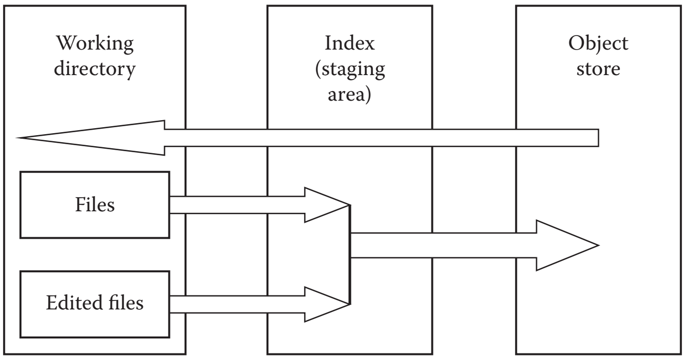


FIGURE 17.8 Working directory, index (staging area), and repository.

17.5.7.4 Directed Acyclic Graphs

In order to plan or visualize the history of a repository structure, a directed acyclic graph (DAG), or *commit graph*, can be used. The name of the graph is derived from the fact that the flow of commits happens along the arrows of the graph (directed), and there is no way you can form a closed circle of commits by following the arrows (it is acyclic). We show an example in Figure 17.9, and will employ this graphic aid to help you visualize the state and the history of the kinds of commits we show.

In Figure 17.9, the circles represent commits, and arrows point from a commit to its parent(s). Time flows from left to right in a DAG, although there is no precise correlation in terms of a time or date stamp on any of the commits. There is just the implication that commits to the left happened earlier than commits to the right in the graph. For most people, this is counterintuitive; usually we see the arrow pointing from something to its successor. In the DAG, arrows point backward toward a parent from a child. The first commit has no parents and is called a *root commit*; it was the initial commit in this repository's history. Most commits have a single parent, indicating that they evolved in a linear way from a single previous state of the project, usually incorporating a set of related changes or edits. A commit that has multiple parents is called a *merge commit*. This indicates that the commit incorporates the changes made on one branch of the commit graph into a commit on another branch of the graph.

There are two other important features of a DAG shown in Figure 17.9. The last commit on the "release" branch has a tag at the top of it, which could contain a descriptive abbreviation of the name of that commit—perhaps "Version 1.0," denoting that this is the first release of the software project. Also, the letter "H" represents the position of HEAD, or the currently checked-out commit on the master branch. We will omit the arrowheads in such diagrams from now on.

The labels on the right side of this picture—**release**, **master**, and **dev**—are the named branches. The branch name refers to the latest commit on that branch. Such a commit is called the *tip of the branch*. The branch itself is defined as the collection of all commits in the graph that are reachable from the tip by following the parent arrows backward along the history to the initial commit.

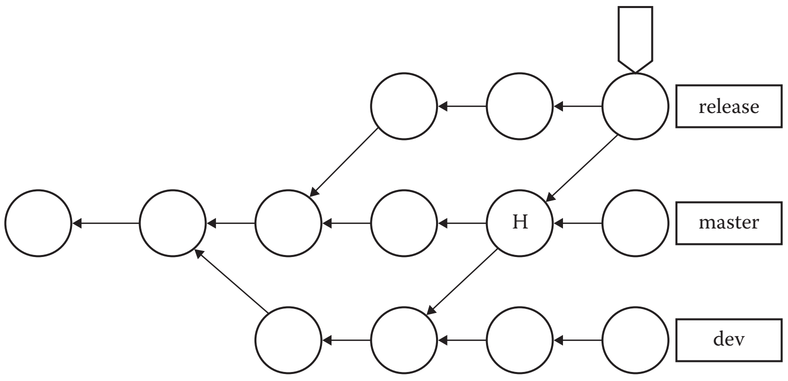


FIGURE 17.9 Example DAG.

17.5.7.5 Contents of the Object Store

Now that we have illustrated how the working directory is structured, how commits are staged, and the general layout of a repository as shown in a DAG, we can now take a finer-grained look at the objects contained in the object store. Figure 17.10 shows the four object types that are found in the object store and the relationships between those objects.

Remember from the DAG shown in Figure 17.9 that time flows from left to right and an arrow points from a child to a parent. Starting at the top, we see a rectangle to the right representing the branch name, and another smaller square representing a tag object. The name by default is **master**, but can be assigned text that is more meaningful. The tag is a shorthand label that might represent the initial release number or version of the software code. The circles represent commits. So this diagram shows two commits. The triangles represent tree objects, which can be thought of as directory or linking information between commits and blobs. Finally, at the bottom are a number of blobs, shown linked to the trees that point to them.

In the next section, we discuss a few examples involving Git and GitHub. Before discussing these examples, we give a brief description of the `git` command.

SYNTAX

```
git [options] [option arguments] <command> [<command arguments>]
```

Purpose: A distributed version control system with a rich command set that tracks the content of changes in source code files

Commonly used commands and command arguments:

`add <file>`	Stage a file to the index
`commit`	Take a snapshot of the working tree
`config --global username "<name>"`	Configure a repository global username
`log --oneline`	Display a short log of commits on the current branch
`init`	Initialize a repository in the current directory
`status`	Display the status of a repository

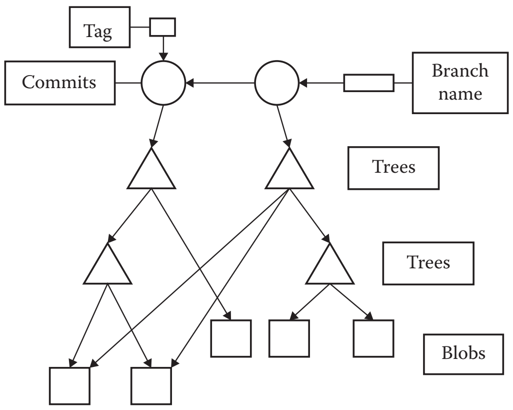


FIGURE 17.10 Contents of the object store.

17.5.7.6 Examples of Using Git and GitHub

We show a few examples describing how to make use of Git and GitHub. Each example consists of the following:

1. Topic covered in the example
2. Objectives of the example
3. Introductory material for understanding the example
4. Git commands used in the example
5. Prerequisites for carrying out the example
6. Detailed procedure
7. Conclusions

Example 17.1: A Short Introduction to Git

Objectives: To briefly illustrate the Git staging model, and how to see differences between the various states of the parts of the repository.

Introduction: As shown previously, the workflow in Git basically follows a pattern of *add*, *edit*, *modify*, *stage*, and *commit*. This is the *staging model*. In this example, we show the essential Git commands that allow you to do that, repetitively if necessary. As you do more commits you add, in a linear fashion, more nodes on the branch named **master** *downstream*, after you create the first node in this example. Earlier commits are called *upstream commits*. The history of the repository flows downstream from the initial commit to the latest commit. Similar to the UNIX `diff` command, the four basic forms of the `git diff` command allow you to examine and compare the files and directories present during different states of the repository.

Git commands referenced: Table 17.1 shows the Git commands, and a brief description of each, that are used in this example. It is arranged in the order presented. Any argument enclosed in < > is a string of text. In order to get a more complete description of all the commands in the table, you can look at the man page for a particular command. For example, `man git-status` gives you a complete man page for the `git status` command.

Prerequisites: The following are the prerequisites for carrying out this example:

1. Having Git installed on your PC-BSD or Solaris system
2. Reading through and doing the in-chapter exercises shown in the previous subsections

Procedure: Do the following steps, in order, to meet the objectives of this example.

1. Create a working directory within which your Git repository will exist, and make that the current working directory.

```
% mkdir short-git
```

TABLE 17.1 Git Commands Referenced

Command	Description
`git config --global user. name "<name>"`	Sets the author of commits in this repository
`git config --global user. Email <email_address>`	Sets the e-mail address of the author of commits in this repository
`git init`	Creates the **.git** directory in the working directory, initializing the data structures and objects necessary for a repository to exist
`git status`	Reports the on the differences between files in the working directory and the index, and what files are untracked.
`git add <file>`	Stages a file to the index
`git commit`	Takes a snapshot of the index, both files and directories
`git diff`	Shows the difference between two project states, in this form meaning your working directory and the index
`git diff <commit_identifier>`	Shows the differences between your working tree and a specifically identified commit
`git diff –cached <commit_identfier>`	Shows the differences between staged files in the index and a specifically identified commit
`git diff <commit_id_1> <commit_id_2>`	Shows the difference between two project states, in this form between commits **commit_id_1** and **commit_id_2**
`git log –oneline`	Shows history of commits in an abbreviated format

```
% cd short-git
%
```

2. Do an initial configuration of Git.

```
% git config --global user.name "your_name"
% git config --global user. Email your_email_address
%
```

3. Initialize a repository in the current working directory.

```
% git init
%
```

This command initializes an empty Git repository in **/usr/home/bob/short-git/.git/**.

4. Create and save a short C program, named **hello.c**, in the current working directory, as shown:

```
% cat hello.c
#include <stdio.h>
int main()
{
      printf("%s\n , "Hello Linus");
      return 0;
}
%
```

5. Compile **hello.c**, and list the files in the current working directory.

```
% gcc47 hello.c
% ls
a.out   hello.c
%
```

6. Use the `git status` command to examine the status of the repository at this point. It will show you that you are on the branch named **master**, you can do your initial commit, the untracked files are **a.out** and **hello.c**, and you can stage those files by using `git add`.

```
% git status
On branch master
Initial commit
Untracked files:
  (use "git add <file>..." to include in what will be
  committed)
        a.out
        hello.c
nothing added to commit but untracked files present (use
"git add" to track)
%
```

7. Stage only the source code file **hello.c** with the `git add` command.

```
% git add hello.c
%
```

8. Now do your initial commit.

```
% git commit
%
```

Use the default text editor to put the message `Added the Hello Linus program` on the first line of the file that appears, and then save and quit that file.

```
/usr/home/bob/short-git/.git/COMMIT_EDITMSG: 13 lines, 285
characters.
[master (root-commit) 0527389] Added the Hello Linus
program
 1 file changed, 6 insertions(+)
 create mode 100644 hello.c
```

9. Look at the status of the repository.

```
% git status
On branch master
Untracked files:
 (use "git add <file>..." to include in what will be
 committed)
        a.out
```

```
nothing added to commit but untracked files present (use
"git add" to track)
%
```

10. Now lets make a change in the file **hello.c**, and track the changes with some of the forms of the `git diff` command. First, edit the file with your favorite text editor, and on the fourth line, add `Torvalds` after the word `Linus`, as shown in the following command output:

```
% nl hello.c
      1 #include <stdio.h>
      2 int main ()
      3 {
      4      printf("%s\n", "Hello Linus Torvalds");
      5      return 0;
      6 }
%
```

11. Examine the status of the repository. The output shows that the file has been changed since the last commit, but has not been staged for a new commit yet.

```
% git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be
  committed)
  (use "git checkout -- <file>..." to discard changes in
  working directory)
        modified:   hello.c
Untracked files:
  (use "git add <file>..." to include in what will be
  committed)
        a.out
no changes added to commit (use "git add" and/or "git
commit -a")
%
```

12. Now we use `git diff` to see the difference between what is in the working directory and the index.

```
% git diff
diff --git a/hello.c b/hello.c
index f907257..621aad0 100644
--- a/hello.c
+++ b/hello.c
@@ -1,6 +1,6 @@
#include <stdio.h>
 int main ()
 {
-     printf("%s\n", "Hello Linus");
+     printf("%s\n", "Hello Linus Torvalds");
      return 0;
 }
%
```

13. Now stage the file **hello.c** in preparation for a new commit.

```
% git add hello.c
%
```

14. If you execute `git diff`, since there are no differences between what is in the working directory and the index, you get no output.

```
% git diff
%
```

15. Now use another form of the command: `git diff –cached commit`, to see the differences between the files staged in the index and any given commit. If you omit the `commit` argument, HEAD is used as the default commit. Remember from the glossary that HEAD is a reference to the current commit. The output should be exactly the same as from step 12. That's because what is in the working directory is the current commit as seen in the object store: HEAD, the current last commit.

```
% git diff --cached
diff --git a/hello.c b/hello.c
index f907257..621aad0 100644
--- a/hello.c
+++ b/hello.c
@@ -1,6 +1,6 @@
 #include <stdio.h>
 int main ()
 {
-     printf("%s\n", "Hello Linus");
+     printf("%s\n", "Hello Linus Torvalds");
      return 0;
 }
%
```

16. Now commit the changes using the `-m` option of `git commit`, which allows you to add a message line right on the command line instead of in an editor.

```
% git commit -m "Added Torvalds to Linus"
[master 37664b9] Added Torvalds to Linus
 1 file changed, 1 insertion(+), 1 deletion(-)
%
```

17. Examine the history of commits with the `git log` command in an abbreviated format with the `–oneline` option.

```
% git log --oneline
37664b9 Added Torvalds to Linus
```

```
0527389 Added the Hello Linus program
%
```

18. See the differences between the two commits we have done so far by using `git diff commit1 commit2`, where **commit1** can be referred to as **0527389**, and **commit2** can be referred to as **37664b9**. These references are seen in the `git log –oneline` command output in step 17.

```
% git diff 0527389 37664b9
diff --git a/hello.c b/hello.c
index f907257..621aad0 100644
--- a/hello.c
+++ b/hello.c
@@ -1,6 +1,6 @@
#include <stdio.h>
int main ()
{
-     printf("%s\n", "Hello Linus");
+     printf("%s\n", "Hello Linus Torvalds");
      return 0;
 }
%
```

19. Examine with `git diff` the differences in only commit **0527389**.

```
% git diff 0527389
diff --git a/hello.c b/hello.c
index f907257..621aad0 100644
--- a/hello.c
+++ b/hello.c
@@ -1,6 +1,6 @@
 #include <stdio.h>
 int main ()
 {
- printf("%s\n", "Hello Linus");
+ printf("%s\n", "Hello Linus Torvalds");
return 0;
}
%
```

20. Repeat steps 4–19 as many times as you want to, each time creating or modifying new or existing files in the working directory. Also, it would be helpful to repeat this entire example several times, in several new directories with newly created repositories to gain practice. Each time, you stage the files with `git add` and then commit the additions or modifications with `git commit`. Then examine the differences as shown with `git diff` and its four variations. As you do more commits you are adding, in a linear fashion, more and more nodes downstream on the branch named **master**.

Conclusions: This short example illustrated the staging model in Git. It introduced a small set of Git commands that allowed you to implement the model, once or repetitively, and see the differences between commits.

EXERCISE 17.12

The `git diff` command is similar to what UNIX command?

Example 17.2: Creating, Editing, and Branching a Git Repository

Objectives: To introduce the Git commands that create, edit, and allow you to develop a C source code project along different branches in a Git repository. To show how different branches may be merged.

Introduction: In order to appreciate and utilize the Git concepts shown in the previous section, we present a complete Git example of repository creation, editing, branching, and merging. Maintaining source code files and their history is the primary objective of Git. In the following example we create C source code files as needed with a text editor in the directory that has the Git repository in it. This method of introducing the files into the working directory does not preclude placing those files in that directory by any other viable means, for example by copying them from another directory or file system. We then edit those files to change their content and commit those changes. Finally, we show how to create branches along which different lines of development of the source code can proceed, and how to merge different branches. We try to emphasize the staging model, or the *edit-stage-commit* workflow model, as detailed in the previous section, throughout this example. We have purposefully not done commits and merges of branches that would produce merge conflicts. The mechanisms and strategies for resolving content conflicts are more usefully covered in other Git reference sources beyond the scope of this section, just as the mechanisms and strategies for resolving context conflicts are.

Git Commands Referenced: Table 17.2 shows the Git commands, and a brief description of each, that are used in this example. It is arranged in the order presented. Any argument enclosed in < > is a string of text. In order to get a more complete description of all the commands in the table, you can look at the man page for a particular command. For example, `man git-status` gives you a complete man page for the `git status` command.

Prerequisites: The following are the prerequisites for carrying out this example:

1. A recent version of Git available on your system, executable by an ordinary user from the command line. In PC-BSD, Git is preinstalled. For Solaris, see the addendum in Section 17.5.7.9 showing how to install Git.
2. Being able to use a text editor, such as vi, vim, or emacs, to create C program source code files.
3. Having reviewed and done the in-chapter exercises in Section 17.1 on Git concepts. This not only gives you a conceptual, top-down view of Git, but also

TABLE 17.2 Git Commands Used

`git init`	Create a Git repository in the current directory
`git status`	View the status of each file in a repository
`git add <file>`	Stage a file for the next commit
`git commit`	Commit the staged files with a descriptive message
`git log`	View a repository's commit history
`git config --global user.name "<name>"`	Define the author name to be used in all repositories
`git config --global user. Email <email>`	Define the author e-mail to be used in all repositories
`git checkout <commit-id>`	Move a previous commit into the working directory
`git tag -a <tag-name> -m "<description>"`	Create an annotated tag pointing to the most recent commit
`git revert <commit-id>`	Undo the specified commit by applying a new commit
`git reset –hard`	Reset tracked files to match the most recent commit
`git clean –f`	Remove untracked files
`git reset --hard / git clean –f`	Permanently undo uncommitted changes
`git branch`	List all branches
`git branch <branch-name>`	Create a new branch using the working directory as its base
`git checkout <branch-name>`	Make the working directory and HEAD match the specified branch
`git merge <branch-name>`	Merge a branch into the checked-out branch
`git branch -d <branch-name>`	Delete a branch
`git rm <file>`	Remove a file from the working directory (if applicable) and stop tracking that file

shows you how to obtain Git help and use the man pages on the system for Git commands.

4. Completion of Example 17.1.

Procedure: Do the steps shown, in the order presented, to meet the objectives of this example. This is a long and detailed example. If you make mistakes, which for a beginner not familiar with the commands are irrevocable, simply start over again in a new directory that has another name than the one shown in step 1.

1. The first step in creating a repository to retain a history of your source code project files is to create a directory within which the repository can exist. We name this directory **first-git**. Then you can do a very elementary configuration of Git to identify yourself to the system.

```
% mkdir first-git
% cd first-git
% git config --global user.name bob
% git config --global user. Email "your_email_address"
%
```

2. Create a C source code file named **first.c** with the text editor of your choice. Save it in the current working directory, which should be **first-git**.

3. The next command initializes the repository, which enables the Git program in the current working directory. There is now a **.git** subdirectory in **first-git** that stores all the tracking data for our repository. The **.git** folder is the only difference between a Git repository and an ordinary folder, so deleting it will turn your project back into an unversioned collection of files.

```
% git init
Initialized empty Git repository in /usr/home/bob/first-
git/.git/
%
```

4. Before we try to start creating revisions, view the status of our new repository. Execute the following command:

```
% git status
On branch master
Initial commit
Untracked files:
  (use "git add <file>..." to include in what will be
  committed)
        first.c

nothing added to commit but untracked files present (use
"git add" to track)
%
```

   This status message tells you that we are about to make our initial commit and that we have nothing to commit but *untracked files*. An untracked file is one that is not under version control. Git doesn't automatically track files because there are often project files that we don't want to keep under revision control. These might be binaries created by a C program, compiled Python modules (**.pyc** files), or any other unnecessary files. To keep a project small and efficient, you should only track source files and omit anything that can be generated from those files. This latter content is part of the build process, not revision control.

5. The next step stages the file **first.c** in preparation for doing the first commit.

```
% git add first.c
%
```

   We added **first.c** to the snapshot of the index for the next commit. Git's term for creating a snapshot is called *staging* because we can add or remove multiple files before actually committing it to the project history. The index holds a snapshot of the content of the working tree, and it is this snapshot that is taken as the contents of the next commit. Thus, after making any changes to the working directory and before running the commit command, you must use the `add` command to add any new or modified files to the index.

6. Now we examine the repository status with the `git status` command.

```
% git status
On branch master
```

```
Initial commit
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   first.c
%
```

Now, instead of **first.c** being an untracked file, it is shown as being staged to be committed.

7. We are ready to commit.

```
% git commit
%
```

The first part of committing is to use the default text editor you are put into by Git to add `The initial commit` as the first line in that file. Then save and quit that file.

```
/usr/home/bob/first-git/.git/COMMIT_EDITMSG: 10 lines, 245
characters.
[master (root-commit) 74088f6] The initial commit
 1 file changed, 1 insertion(+)
 create mode 100644 first.c
```

8. We need a new command, `git log`, to view the project revision history. When you execute this command, Git will output information about our first commit:

```
% git log
commit 74088f645993f3df16f27565628ea38c271357e0
Author: bob <your_email_address>
Date:   Mon Nov 10 18:59:44 2014 -0800
    The initial commit
%
```

9. We continue to add new C source code files to our working directory. Create two C source code files named **second.c** and **third.c** with the text editor of your choice. Save them in the current working directory, which should be **first-git**.
10. We now need to stage those two new files, in preparation for committing them to our repository.

```
% git add second.c third.c
%
```

11. Take a look at the status of the repository.

```
% git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
        new file:   second.c
        new file:   third.c
%
```

12. Take a look at the history of the repository.

```
% git log
commit 74088f645993f3df16f27565628ea38c271357e0
Author: bob <your_email_address>
Date:   Mon Nov 10 18:59:44 2014 -0800
    The initial commit
%
```

13. Commit the two new files.

```
% git commit
%
```

Use the text editor that automatically launches to add `second.c and third.c added` as the first line in the file. Then save the file and quit the text editor.

```
/usr/home/bob/first-git/.git/COMMIT_EDITMSG: 8 lines, 255
characters.
[master 4ea0534] second.c and third.c added
 2 files changed, 2 insertions(+)
 create mode 100644 second.c
 create mode 100644 third.c
```

14. The `git add` command is used to stage new files. It can also be used to stage modified files. In order to demonstrate this, use a text editor to modify the previously created C source code files **first.c**, **second.c**, and **third.c**.
15. Then take a look at the status of the repository. Git lists the tracked files as being modified.

```
% git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be
committed)
  (use "git checkout -- <file>..." to discard changes in
  working directory)
        modified:   first.c
        modified:   second.c
        modified:   third.c
no changes added to commit (use "git add" and/or "git
commit -a")
%
```

16. Stage those modified files.

```
% git add first.c second.c third.c
%
```

17. Now commit the modified, staged files. The `-m` option of `commit` lets you specify a commit message on the command line instead of opening a text editor. This is a shortcut method; it has the exact same effect as our previous commits.

```
% git commit -m "Revised all three files"
[master b0a6b40] Revised all three files
 3 files changed, 3 insertions(+)
%
```

18. Our history can now be shown as follows:

```
% git log --oneline
b0a6b40 Revised all three files
4ea0534 second.c and third.c added
74088f6 The initial commit
%
```

    The `git log` command comes with formatting options. For now we show the `--oneline` flag, as in `git log –oneline`.

19. Condensing output to a single line is one way to get an overview of a repository. Another useful configuration is to pass a filename to `git log`:

```
% git log --oneline second.c
b0a6b40 Revised all three files
4ea0534 second.c and third.c added
%
```

20. So far, we have recorded versions of a project into a Git repository. Maintaining these copies provides us with backups. More importantly, we can have independent versions of the state of the project that can be used for the purposes of creating multiple lines or tracks of development. Our next objective is to be able to view the previous states of a project, revert back to them, and reset uncommitted changes if necessary. First, let's return to the state of the repository at the commit **4ea0534 second.c and third.c added**. The HEAD is now at **4ea0534**. The `git checkout` command positions HEAD at any commit we desire, going all the way back to the initial commit.

```
% git checkout 4ea0534
Note: checking out '4ea0534'.
%
```

    You are in the *detached HEAD* state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

    If you want to create a new branch to retain commits you create, you may do so (now or later) by using `-b` with the `checkout` command again. For example:

```
% git checkout -b new_branch_name
HEAD is now at 4ea0534... second.c and third.c added
%
```

21. Let's go back to the initial commit.

```
% git checkout 74088f6
Previous HEAD position was 4ea0534... second.c and third.c
added
```

```
HEAD is now at 74088f6... The initial commit
%
```

22. We can check the status of the repository at this point.

```
% git status
HEAD detached at 74088f6
nothing to commit, working directory clean
%
```

23. All previous steps worked on the master branch, where our second and third commits reside. To retrieve our complete history, we just have to check out this entire branch. This is a very brief introduction to branches, but it's all we need to know to navigate between commits. The following command makes Git update our working directory to reflect the state of the master branch's snapshot. It recreates the **second.c** and **third.c** files for us, and the content of **first.c** is updated as well. We're now back to the current state of the entire commits history of the project.

```
% git checkout master
Previous HEAD position was 74088f6... The initial commit
Switched to branch 'master'
%
```

24. Tags are references to milestones, or releases in a software project. They let developers easily browse and check out important revisions. For example, we can now use the **v1.0** tag to refer to the third commit instead of its random ID. To view a list of existing tags, execute `git tag` without any arguments. So, we can label this a stable version of the C program modules. The `-a` option tells Git to create an annotated tag, which lets us record our name, the date, and a descriptive message, specified via the `-m` option. We can finalize it by tagging the most recent commit with a version number as follows:

```
% git tag -a v1.0 -m "Stable version of the software"
%
```

25. Now we can add C modules to the working directory that allow us to experiment, without committing those modules. Use the text editor of your choice to create an experimental C source code file named **experiment.c**, and save it in the current working directory.
26. Then stage that file.

```
% git add experiment.c
%
```

27. Check on the status of the repository.

```
% git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
       new file: experiment.c
%
```

28. Commit that file.

```
% git commit -m "Add an experimental C program"
[master 1707e3e] Add an experimental C program
 1 file changed, 1 insertion(+)
 create mode 100644 experiment.c
%
```

29. See a history of commits to the repository.

```
% git log
commit 1707e3e6f3e12ef1eaa20417dd601bb11180d731
Author: bob <your_email_address>
Date:   Mon Nov 10 19:32:31 2014 -0800
Add an experimental C program
commit b0a6b40c4ab515fb10c036b0d7d2ebcacd281865
Author: bob <your_email_address>
Date:   Mon Nov 10 19:16:05 2014 -0800
Revised all three files
commit 4ea0534d1e88d297d38a0c63e31dab8b6da7c5ca
Author: bob <your_email_address>
Date:   Mon Nov 10 19:08:00 2014 -0800
second.c and third.c added
commit 74088f645993f3df16f27565628ea38c271357e0
Author: bob <your_email_address>
Date:   Mon Nov 10 18:59:44 2014 -0800
The initial commit
%
```

30. Let's go back to our stable revision. Remember that the **v1.0** tag is now a shortcut to the third commit's ID.

```
% git checkout v1.0
Note: checking out 'v1.0'.
%
```

You are in *detached HEAD* state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may do so (now or later) by using -b with the checkout command again. For example:

```
% git checkout -b new_branch_name
HEAD is now at b0a6b40... Revised all three files
%
```

31. After seeing the stable version of the site, we decide to scrap the C code experiment we started in step 25. But, before we undo the changes to the repository, we need to return to the **master** branch. If we didn't, all of our updates would

be on some nonexistent branch. You should never make changes directly to a previous revision.

```
% git checkout master
Previous HEAD position was b0a6b40... Revised all three
files
Switched to branch 'master'
%
```

32. Now examine the history of our repository with the `git log` command. This gives us the shorthand name of the last commit we executed entitled `Add an experimental C program`.

```
% git log --oneline
1707e3e Add an experimental C program
b0a6b40 Revised all three files
4ea0534 second.c and third.c added
74088f6 The initial commit
%
```

33. Now we want to restore our stable release by removing the most recent commit. Make sure to change **1707e3e** to the ID supplied by your system's Git for the experimental C code commit before running the next command. Also, the command we use, `git revert`, undoes the commit we specify as its argument.

```
% git revert 1707e3e
%
```

You are put into the default text editor, which allows you to change the title of the reverted commit. Leave the commit title the same, save the file and quit the editor.

34. Look at what files are in the working directory, and see a history of your commits.

```
% ls
first.c second.c third.c
% git log --oneline
03ece49 Revert "Add an experimental C program"
1707e3e Add an experimental C program
b0a6b40 Revised all three files
4ea0534 second.c and third.c added
74088f6 The initial commit
%
```

Notice that instead of deleting the `Add an experimental C program` commit, Git undoes the changes it contains, then adds on another commit showing the reversion. So, our fifth commit and our third commit represent the exact same snapshot, as follows. Again, Git is designed to never lose history: the fourth snapshot is still accessible, just in case we want to continue developing it.

35. Now we can try to add a file that we definitely will want to get rid of completely. Use your text editor to create a file named **dumbc.c**, and then edit **first.c** to make a small change in it. Now look at the status of the repository.

```
% git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be
committed)
  (use "git checkout -- <file>..." to discard changes in
  working directory)
        modified:   first.c
Untracked files:
  (use "git add <file>..." to include in what will be
  committed)
        dumbc.c
no changes added to commit (use "git add" and/or "git
commit -a")
%
```

36. We have a tracked file and an untracked file that need to be changed. First, we'll take care of the tracked **first.c**.

```
% git reset --hard
HEAD is now at 03ece49 Revert "Add an experimental C
program"
%
```

This changes all tracked files to match the most recent commit. You can also pass a filename to this command to reset only that file—for example, `git reset --hard first.c`. The `--hard` flag is what actually updates the file. Running `git reset first.c` without any flags will simply unstage the file, leaving its contents as is. In either case, `git reset` only operates on the working directory and the staging area, so our `git log` history remains unchanged.

37. Now remove the **dumb.c** file. Of course, we could manually delete it, but using Git to reset changes eliminates human errors when working with several files in large teams. Run the following command:

```
% git clean -f
Removing dumbc.c
%
```

This will remove all untracked files. With **dumb.c** gone, `git status` should now tell us that we have a "clean" working directory, meaning our project repository matches the most recent commit. Be careful with `git reset` and `git clean`. Both operate on the working directory, not on the committed snapshots. Unlike `git revert`, they permanently undo changes, so make sure you really want to delete what you're working on before you use them.

38. To begin creating and using branches, list what branches exist at this point.

```
% git branch
* master
%
```

This command displays the only current branch, named * **master**. The **master** branch is Git's default branch, and the asterisk next to it means that it is currently checked out. This means that the most recent snapshot in the master branch resides in the working directory. There is only one working directory for each project, only one branch can be checked out at a time.

39. Look at some previous commits before we begin creating a new branch. First get a shorthand list of the repository commit history.

```
% git log --oneline
03ece49 Revert "Add an experimental C program"
1707e3e Add an experimental C program
b0a6b40 Revised all three files
4ea0534 second.c and third.c added
74088f6 The initial commit
%
```

40. Next, checkout the **Add an experimental C program** commit.

```
% git checkout 1707e3e
Note: checking out '1707e3e'.
%
```

You are in *detached HEAD* state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may do so (now or later) by using `-b` with the `checkout` command again. For example:

```
% git checkout -b new_branch_name
HEAD is now at 1707e3e... Add an experimental C program
%
```

The HEAD normally is on the tip of a development branch, meaning you are on that branch. But when we checked out the previous commit, the HEAD moved to the middle of the branch. We are no longer on the master branch since it contains more recent snapshots than the HEAD. This is reflected in the Git branch output from the previous command, which tells us that we're currently not on a branch.

41. We can now create a branch from this commit. Name it **test**.

```
% git branch test
%
```

42. To be able to add commits to the new branch, move onto that branch by checking it out.

```
% git checkout test
```

```
Switched to branch 'test'
%
```

43. Use your favorite text editor to make minor changes to the file **experiment.c**, so that we can begin development along this branch. Be sure to save the modified **experiment.c**.
44. Now stage the modified **experiment.c**.

```
% git add experiment.c
% git status
On branch test
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
        modified:   experiment.c
%
```

45. The following commit will create a fork in our project repository, as shown in Figure 17.11.

```
% git commit -m "Modified experiment.c"
[test 1ff0291] Modified experiment.c
 1 file changed, 1 insertion(+)
%
```

46. Take a look at the history of commits in abbreviated form.

```
% git log --oneline
1ff0291 Modified experiment.c
1707e3e Add an experimental C program
b0a6b40 Revised all three files
4ea0534 second.c and third.c added
74088f6 The initial commit
%
```

The history before the fork is considered part of the new branch. So, since we are on the branch test, the test history spans all the way back to the first commit. The project repository has a complex history, but each individual branch still has a linear history. Snapshots and commits occur one after another in a linear fashion. This means that we can work within branches in the same way we did in steps 1–37.

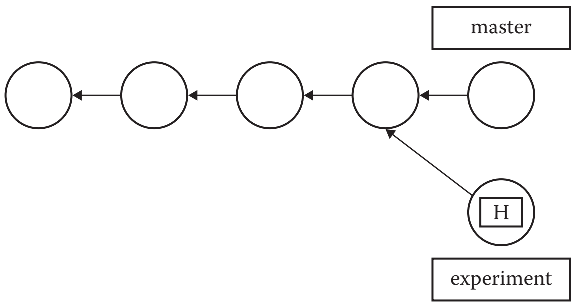


FIGURE 17.11 Forked project repository.

47. Let's add one more snapshot to the test branch. Rename **experiment.c** to **experiment2.c**, then use the following Git commands to update the repository.

```
% mv experiment.c experiment2.c
% git status
On branch test
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be
committed)
  (use "git checkout -- <file>..." to discard changes in
working directory)
        deleted:    experiment.c
Untracked files:
  (use "git add <file>..." to include in what will be
committed)
        experiment2.c
no changes added to commit (use "git add" and/or "git
commit -a")
% git rm experiment.c
rm 'experiment.c'
% git status
On branch test
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
        deleted:    experiment.c
Untracked files:
  (use "git add <file>..." to include in what will be
  committed)
        experiment2.c
% git add experiment2.c
% git status
On branch test
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
        renamed:    experiment.c -> experiment2.c
%
```

The `git rm` command tells Git to stop tracking **experiment.c** (and delete it if necessary), and `git add` starts tracking **experiment2.c**. The `renamed: experiment.c => experiment2.c` message in the final status output shows us that Git knows when we are just renaming a file. You could have just made editing changes to **experiment.c** to justify moving the branch forward with another commit. Our snapshot is staged and ready to be committed.

48. We now do a new commit along the new branch.

```
% git commit -m "Renamed experiment.c to experiment2.c"
[test 27d1e0d] Renamed experiment.c to experiment2.c
 1 file changed, 0 insertions(+), 0 deletions(-)
```

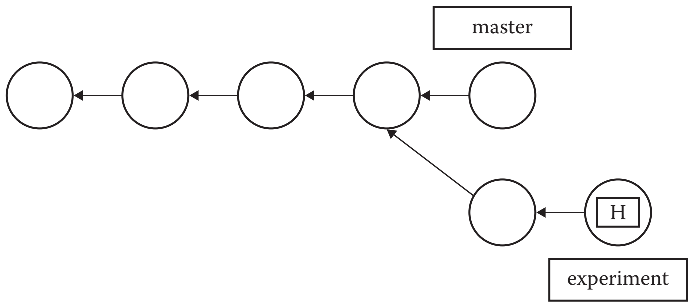


FIGURE 17.12 DAG of project repository.

```
 renamed experiment.c => experiment2.c (100%)
%
```

49. Look at the history of commits along this branch. Our project repository now looks as shown in Figure 17.12.

```
% git log --oneline
27d1e0d Renamed experiment.c to experiment2.c
1ff0291 Modified experiment.c
1707e3e Add an experimental C program
b0a6b40 Revised all three files
4ea0534 second.c and third.c added
74088f6 The initial commit
%
```

50. Now we will fork another branch off the master branch. In preparation for doing this, return the HEAD to the **master** branch by using the `git checkout` command.

```
% git checkout master
Switched to branch 'master'
%
```

51. Now that you are back on the **master** branch, list the branches in this repository.

```
% git branch
* master
  test
% git log --oneline
03ece49 Revert "Add an experimental C program"
1707e3e Add an experimental C program
b0a6b40 Revised all three files
4ea0534 second.c and third.c added
74088f6 The initial commit
%
```

52. We will now create a new branch, forked off the **master** branch and named **modules**.

```
% git branch modules
%
```

53. Make the modules branch the current branch.

```
% git checkout modules
Switched to branch 'modules'
%
```

54. In your favorite text editor, create a C program file called **module1.c**. Then stage **module1.c**.

```
% git add module1.c
%
```

55. Check the status of the repository.

```
% git status
On branch modules
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
        new file:   module1.c
%
```

56. Now commit **module1.c**.

```
% git commit -m "Added module1.c"
[modules 9b37c23] Added module1.c
 1 file changed, 1 insertion(+)
 create mode 100644 module1.c
%
```

57. In your favorite text editor, add a reference to the C code in **module1.c** in the files **first.c**, **second.c**, and **third.c**.
58. Stage the changes you made in **first.c**, **second.c**, and **third.c**.

```
% git add first.c second.c third.c
%
```

59. Check the status of the project repository.

```
% git status
On branch modules
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
        modified:   first.c
        modified:   second.c
        modified:   third.c
%
```

60. Commit the changes to the files **first.c**, **second.c**, and **third.c**.

```
% git commit -m "Add references to module1.c"
[modules 436a966] Add references to module1.c
 3 files changed, 3 insertions(+)
%
```

61. Now examine the history of commits along this branch.

```
% git log --oneline
436a966 Add references to module1.c
```

```
9b37c23 Added module1.c
03ece49 Revert "Add an experimental C program"
1707e3e Add an experimental C program
b0a6b40 Revised all three files
4ea0534 second.c and third.c added
74088f6 The initial commit
%
```

A DAG of your project repository at this point is shown in Figure 17.13.

62. In preparation for merging the **modules** branch with the **master** branch, do the following. First, switch to the **master** branch, then check the files in the working directory. Finally look at the commit history of the repository as seen along the **master** branch.

```
% git checkout master
Switched to branch 'master'
% ls
first.c second.c third.c
% git log --oneline
03ece49 Revert "Add an experimental C program"
1707e3e Add an experimental C program
b0a6b40 Revised all three files
4ea0534 second.c and third.c added
74088f6 The initial commit
%
```

63. We will now merge the **modules** branch with the **master** branch. This command always merges into the current branch. The **modules** branch is unchanged. Check the history of commits with `git log --oneline` to make sure that **modules**'s history of commits has been added to **master**'s history of commits. The DAG representing the final state of the repository is shown in Figure 17.14.

```
% git merge modules
Updating 03ece49..436a966
```

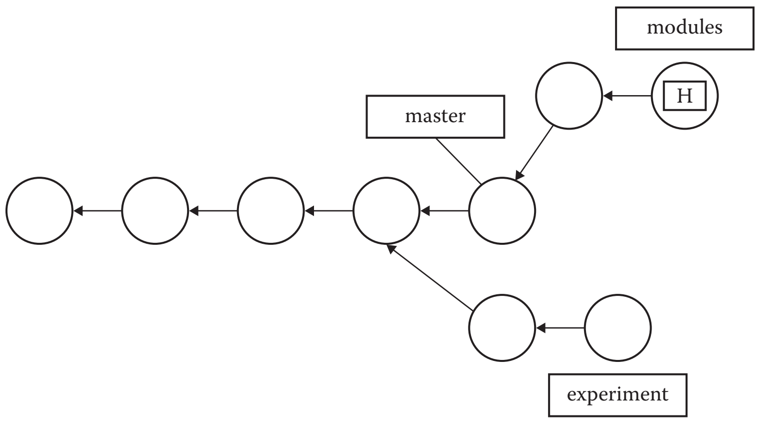


FIGURE 17.13 DAG of repository with three branches.

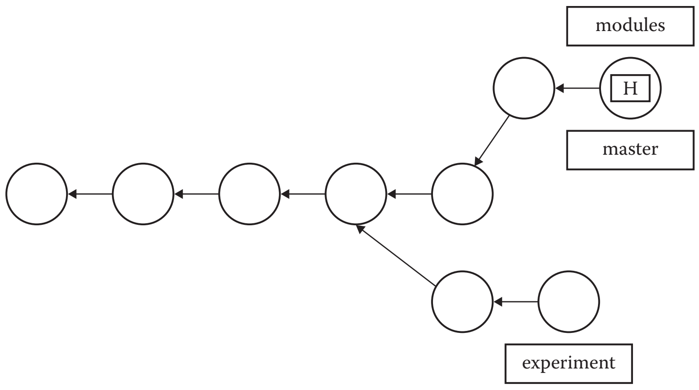


FIGURE 17.14 DAG of final state of project repository.

```
Fast-forward
 first.c   | 1 +
 module1.c | 1 +
 second.c  | 1 +
 third.c   | 1 +
 4 files changed, 4 insertions(+)
 create mode 100644 module1.c
% git log --oneline
436a966 Add references to module1.c
9b37c23 Added module1.c
03ece49 Revert "Add an experimental C program"
1707e3e Add an experimental C program
b0a6b40 Revised all three files
4ea0534 second.c and third.c added
74088f6 The initial commit
%
```

Conclusions: In this example, we created C source code files as needed with a text editor in the directory that has the Git repository in it. We then edited those files to change their content and committed those changes. Finally, we showed how to create branches along which different lines of development of the source code can proceed, and how to merge different branches. We emphasized the staging model, or edit-stage-commit workflow model, as detailed in the previous section, throughout this example.

EXERCISE 17.13

Under what circumstances would you want to track, or stage, other kinds of files in a Git repository related to the C program development and build process?

EXERCISE 17.14

If, after step 63, you were to create a new text file in the working directory, but not stage and commit, would that file still be in the working directory after you do a checkout of the initial commit? Why/why not?

17.5.7.7 GitHub as a Remote Repository

GitHub is a popular remote repository where you can easily and securely work together with a team to do the development and maintenance of a software project. We first provide some background information on Git URLs and *refspecs*. Then, in this section's examples, we show the basics of how to take files from a local repository and put them on a GitHub repository using the UNIX command line. We also show how to take files from GitHub and retrieve them back on to a local repository. The basis and groundwork for these operations are expedited on the UNIX command line with the Git commands we have illustrated thus far. Therefore, having completed the previous subsections on Git is necessary to your understanding of the GitHub interactions shown here. We also introduce new Git commands to allow you to work with a GitHub remote repository.

We do not show how to get an account on GitHub, or how to create a new repository using the web-based GUI interface of GitHub. It is assumed in all of the examples that you can do those two basic steps via a Web browser at www.github.com, and can navigate to the URL we show. The repository on your system is called the *local* or *current repository*, and a repository like GitHub is called the *remote repository*.

Use the `git remote` command and its options and arguments to create, remove, manipulate, and view a remote. For example, to add a remote reference specification, or refspec, to the current local repository, use the `git remote add` command with options and arguments. You can also look at what has been defined as a remote in the **.git/config** file. All the remotes you added are recorded in the **.git/config** file and can be manipulated using `git config` and its options and arguments.

The basic Git commands that refer to remote repositories are

`git clone`	Transfers a remote repository into the local repository
`git fetch`	Retrieves objects and their related data structures from a remote repository
`git pull`	Merges changes from a remote repository into a corresponding local branch
`git push`	Copies objects and their related data structures to a remote repository
`git ls-remote`	Lists references in a given remote repository

17.5.7.7.1 Git URLs For the `git remote` command, Git names the argument forms of reference to the remote repository as *Uniform Resource Locators* (URLs). A Git URL that refers to a repository on a local file system can be:

```
/pathname/repo.git
file:///pathname/repo.git
```

The first reference form uses hard links within the UNIX file system to directly share exactly the same objects between the current and remote repository. The second and preferred form copies the data instead of sharing it via links.

A Git URL that refers to a repository on a remote system can take several forms. These forms include http, https, ssh, scp, rsync, and ftp. The primary and preferred ways of designating a remote repository using http or https, and which we use in the examples, are as follows.

```
http://github.com/pathname/repository_name
https://github.com/pathname/repository_name
```

where **pathname** is a username on GitHub, and **repository_name** is a specific named repository for that user. The named repository does not have to end in with a **.git** suffix. These two URL forms are most favored by GitHub.

Server firewalls usually allow the http port 80 and https port 443 to remain open, and by default on PC-BSD, port 22 for ssh.

For a remote repository whose data must be retrieved across a wide area network, such as the Internet, you can also use the Git native protocol, which refers to the custom protocol used internally by Git to transfer data. Examples of a native protocol URL include

```
git://example.com/pathname/repo.git
git://example.com/~user/pathname/repo.git
```

These forms are used by Git to publish repositories for anonymous read. You can both clone and fetch using these URL forms.

The Git native protocol can be tunneled over an ssh connection using the following URL specification:

```
ssh://[user@]github.com[:port]/pathname/repo.git
```

where **user@** is the client-side ssh username on the local system, **port** is the optional designation of a port on the client other than the default port 22, **pathname** is the username on GitHub, and **repo.git** is the name of the specific repository on GitHub.

Git also supports a URL form with scp-like syntax. It is identical to the ssh forms, but there is no way to specify a port parameter.

```
[user@]example.com:/pathname/repo.git
```

For a more complete list and explanation of the remote URL specifications, use the command `man git-clone`. Then, you should page down to the URL specifications section of the man page.

17.5.7.7.2 Understanding Remote Pull and Push Operations Git and GitHub workflow models, branching strategies, and particularly the resolution of merge conflicts when working with those models and strategies, can be very complex. Those models are also as varied as the different kinds of software development and maintenance teams, the size of those teams, and their respective goals. For the basic Git commands that allow you to work with remote repositories, it is helpful for a beginner to know some background material for those commands discussed at the beginning of Section 17.5.7. Since most of your workflow as a beginner involves using the `git push` and `git pull` commands, it is very helpful to know what the underlying assumptions and basis for those commands are.

After you have cloned a remote repository to a local one, the commands `git pull` and `git push` keep the two repositories synchronized as far as their content is concerned. The most important thing to remember about keeping repositories synchronized is that, with regard to content, a repository consists of two things: an object store and a set of references, or *refs*—in other words, a commit graph and a set of branch names and tags that designate commits. When you clone a repository, such as with the command `git clone URL/repository`, the Git does the following things in the order shown:

- Creates a new local repository that is essentially a replica of the remote repository
- Adds a remote named **origin** to refer to the repository being cloned in **.git/config**:

```
[remote "origin"]
fetch = +refs/heads/*:refs/remotes/origin/*
url = URL/repository
```

The line in the **config** file with `fetch` in it is the refspec, an assignment statement that specifies a correspondence between sets of refs in the two repositories: the pattern on the left side of the colon names refs in the remote, associated with the pattern on the right side of the colon, which are the corresponding refs in the local repository.

- Runs the command `git fetch origin`, which updates our local refs for the remote's branches (creating them in this case), and asks the remote to send any objects we need to complete the history for those refs (in the case of this new repository, all of them).
- Checks out the remote repository's current branch (its HEAD ref), giving you a working directory, and **.git** directory in it—that is, a replicated local repository cloned from the remote repository.

So now you can execute the `git show-ref` command as follows and view the local repository refs:

```
% git show-ref --abbrev master
b5216a81 refs/heads/master
b5416a91 refs/remotes/origin/master
%
```

When you give the `git pull` command, Git first executes a fetch on the remote for the current branch, updating the remote's local tracking refs and obtaining any new objects needed to complete the history of those refs—that is, all commits, tags, trees, and blobs reachable from the new branch tips. Then it tries to update the current local branch to match the corresponding branch in the remote. If only one side has added content to the branch, then this will succeed, and is called a *fast-forward update* since one ref is simply moved forward along the branch to catch up with the other.

If both sides have committed to the branch, however, then Git has to do something to incorporate both versions of the branch history into one shared version. By default, this is a merge: Git merges the remote branch into the local one, producing a new commit that refers to both sides of the history via its parent pointers. And this would most likely lead to merge conflicts.

When you give the `git push` command, Git updates the corresponding branch in the remote with your local repository branch contents, sending any objects the remote needs to complete the new state of the remote repository. This will fail if the update is non-fast-forward, and Git will suggest that you first `git pull` in order to resolve the differences between repositories.

Nothing in remote-tracking branches ties the things you do to your repository to the remote; the relationship is one way. Each remote-tracking branch is just a branch in your repository like any other branch, a ref pointing to a particular commit. They are only "remote" in the sense that they point to a remote repository. They track the state of corresponding branches in the remote, and you can update them using the command `git pull`.

A repository can have many remotes, set up at any time; see the `git remote add` command in Example 17.3. If the original repository you cloned from is no longer available, you can fix its URL by editing the **.git/config** file for a particular local repository. You can remove a remote reference entirely with `git remote rm`. This command will remove the remote-tracking branches for that remote repository too.

17.5.7.8 GitHub Examples

The following section illustrates your basic interaction with GitHub as a remote repository, using Git commands from the UNIX command line.

Example 17.3: Basic GitHub Operations

Objective: To create a new repository in your existing account at GitHub, and transmit files to the new GitHub repository from a local repository.

Introduction: In this example, we first create a working directory with a new Git repository in it. Then we add a file to this new local repository and use the `git push` command to move that file up to a repository at GitHub.

Git Commands Referenced: Table 17.3 shows the Git commands, and a brief description of each, that are used in this example. It is arranged in the order presented. Any argument enclosed in < > is a string of text. In order to get a more complete description of all the commands in the table, you can look at the man page for a particular command. For example, `man git-push` gives you a complete man page for the `git push` command.

TABLE 17.3 git Commands

Command	Description
`git init`	Add a Git repository in the current directory
`git add <file>`	Stage `<file>` for the next commit
`git commit -m "Message"`	Execute a commit with `Message` automatically added using the `–m` option.
`git remote add origin <path>`	Identify the valid `<path>` as a Git remote repository reference
`git push -u origin master`	Transmit the branch named **`master`** to the current remote repository and add upstream tracking information
`git remote –v`	List the remotes defined for this repository

Prerequisites: The following are the prerequisites for carrying out this example:

1. Having an account on GitHub and knowing how to add a new repository to that account
2. Having completed the previous subsections that familiarize you with Git commands executed on the UNIX command line
3. Having an Internet connection and a suitable Web browser installed and operating on your UNIX system
4. Local and remote GitHub repositories with only one branch each
5. Having completed Example 17.2

Procedure: Do the following steps, in the order presented, to meet the objectives of this example.

1. Create the working directory and make it the current directory.

```
% mkdir githubtest
% cd githubtest
%
```

2. Add a new file to the directory.

```
% touch README.md
%
```

3. Initialize Git in this new directory.

```
% git init
Initialized empty Git repository in /usr/home/bob/
githubtest/.git/
%
```

4. Stage the file **README.md**, and do an initial commit to the repository.

```
% git add README.md
% git commit -m "First Commit"
[master (root-commit) f971b1d] First Commit
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 README.md
%
```

5. In your Web browser, navigate to GitHub at www.github.com, log in, and create a new repository in your GitHub account. Name that repository **test**.
6. Use the `git remote` command to designate your GitHub repository **test** as a remote repository for the local repository we created in steps 1-4. To find the URL to designate as the GitHub repository, look in the URL bar of your browser when you are in the GitHub repository named **test** that you created in step 5. On our system, in our browser, the URL to this new repository on GitHub is **https://github.com/bobk48/test.git**.

```
% git remote add origin https://github.com/bobk48/test.git
%
```

7. Use the `git push` command to take the local repository and move it up to GitHub. Supply the username and password for the GitHub repository as needed.

```
% git push -u origin master
Username for 'https://github.com': bobk48
Password for 'https://bobk48@github.com': xxx
Counting objects: 3, done.
Writing objects: 100% (3/3), 207 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/bobk48/test.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from
origin.
%
```

8. Check the names of the files in the local repository.

```
% ls
README.md
%
```

   In your browser, check the content of the repository **test**.
9. Use the `git remote` command to list the remote repositories for this local repository.

```
% git remote -v
origin https://github.com/bobk48/test.git (fetch)
origin https://github.com/bobk48/test.git (push)
%
```

   What this output shows you is that you can transfer (using the `git push` command) new content to the remote repository, and get content from it (using the `git fetch` command).
10. To add a new file to the GitHub repository from the local repository, first create the file with your favorite text editor.

```
% vi newfile.txt
... Create and save a new textfile named newfile.txt ...
```

```
newfile.txt: new file: 1 lines, 47 characters.
%
```

11. List the files in your working directory.

```
% ls
README.md newfile.txt
%
```

12. Stage **newfile.txt** and commit it.

```
% git add newfile.txt
% git commit -m "second new file added"
[master 4dc2de7] second new file added
 1 file changed, 1 insertion(+)
 create mode 100644 newfile.txt
%
```

13. Now use `git push` again to push the contents of the repository to your GitHub repository.

```
% git push origin master
Username for 'https://github.com': bobk48
Password for 'https://bobk48@gmail.com@github.com': xxx
Counting objects: 4, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 325 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/bobk48/test.git
   f971b1d..4dc2de7 master -> master
%
```

In your browser, check the content of your GitHub repository named **test**. You should see the same files there that are in your local repository along the branch **master**.

Conclusions: By designating a GitHub repository as a remote repository reference using an https URL, we accomplished moving files up from a local repository to a GitHub repository.

EXERCISE 17.15

What refspec URL and fetch assignments are listed for the repository **test**? What branch refspecs are listed? How did you find this out?

EXERCISE 17.16

What Git command do you use locally to put an earlier, or *upstream*, commit into the working directory?

Example 17.4: Cloning a GitHub Repository

Objectives: To clone an existing remote GitHub repository into a new local repository.

Introduction: In order to share the contents of an existing GitHub repository between members of a software development and maintenance team, it is a usual practice to clone, or copy, a complete repository from GitHub into a new local repository. In the previous Example 17.3, we first created a working directory and a new Git repository in it. Then we added a file to this new local repository and used the `git push` command to move that file up to an existing repository at GitHub. In this example, we will use the `git clone` command to create an entirely new local Git repository from an existing remote GitHub repository. Then we will add a new file to the local repository and use `git push` to transfer that file to the GitHub repository. To simplify things for the beginner, there is only one branch on the remote GitHub repository.

Git Commands Referenced: Table 17.4 shows the Git commands, and a brief description of each, that are used in this example. It is arranged in the order presented. Any argument enclosed in < > is a string of text. In order to get a more complete description of all the commands in the table, you can look at the man page for a particular command. For example, `man git-clone` gives you a complete man page for the `git clone` command.

Prerequisites: The following are the prerequisites for carrying out this example:

1. Having completed the previous subsections that familiarize you with Git commands executed on the UNIX command line
2. Having an Internet connection and a suitable Web browser installed and operating on your UNIX system
3. Having completed Example 17.3 and having your Web browser pointed at the test repository created in that example so you can check on its contents
4. Having access to an account on GitHub that has in it the existing repository **test** created in Example 17.3.

TABLE 17.4 Git Commands Referenced

Command	**Description**
`git clone <remote_designation>`	Transfers a complete repository from the remote designated into a local repository, maintaining the branch and file structure
`git status`	Shows the current state of the repository
`git add <object(s)>`	Stages the named object(s) to the index
`git commit -m "Message"`	Commits the contents of the staged files in the index
`git push <remote_designation> master`	Transfers the working directory to the remote designated on the branch **master**

Procedure: Perform the following steps, in the order presented, to meet the objectives of this example:

1. Create a new empty directory beneath your home directory on your UNIX system named **github_clone** and make that directory the current working directory.

   ```
   % mkdir github_clone
   % cd github_clone
   %
   ```

   This new directory will serve as the file system *landing zone*, within which the `git clone` command shown in the next step will replicate the entire remote GitHub repository.
2. Use the `git clone` command to transfer the contents of the remote GitHub repository into the current working directory. Remember that the URL we show in the command is different from the one that you will be seeing on your system, so make the appropriate changes.

   ```
   % git clone https://github.com/bobk48/test
   Cloning into 'test'...
   remote: Counting objects: 6, done.
   remote: Compressing objects: 100% (3/3), done.
   remote: Total 6 (delta 0), reused 6 (delta 0)
   Receiving objects: 100% (6/6), done.
   Checking connectivity... done.
   %
   ```

3. List the contents of the working directory. It should contain the complete repository from your GitHub repository in Example 17.3. The directory listed is the working directory for the cloned repository. If you descend into that directory, everything that is in your GitHub repository is in **test**.

   ```
   % ls
   test
   %
   ```

4. Make the directory **test** the current directory, in preparation for putting a new file in it that you will then transfer up to the GitHub repository.

   ```
   % cd test
   %
   ```

   The test directory is now your working directory in Git terminology.
5. With your favorite text editor, create a new file, with any contents you want in it, in the directory **test**. Save the file and exit the text editor.

   ```
   % vi newfile2.txt
   newfile2.txt: new file: 1 lines, 58 characters.
   %
   ```

6. List the contents of directory **test**.

```
% ls
README.md    newfile.txt  newfile2.txt
%
```

7. Check the status of the local repository with the `git status` command.

```
% git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be
  committed)
        newfile2.txt
nothing added to commit but untracked files present (use
"git add" to track)
%
```

8. Stage and commit the new file to the local repository, in preparation for transferring it up to GitHub.

```
% git add newfile2.txt
% git commit -m "Added newfile2.txt to test"
%
```

9. Use `git push` to transfer the new file up to the GitHub repository named **test** on the branch **master**.

```
% git push https://github.com/bobk48/test master
Username for 'https://github.com': your_username
Password for 'https://bobk48@gmail.com@github.com': xxx
Counting objects: 4, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 371 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/bobk48/test
   4dc2de7..b651617 master -> master
%
```

10. From your Web browser, examine the GitHub repository named **test**. It should now contain the file you pushed to it in step 9.

Conclusions: The easiest way to create a local repository that is an exact copy of a GitHub repository is to use the `git clone` command.

EXERCISE 17.17

What refspec URL and fetch assignments are listed for the repository **test**? What branch refspecs are listed? How did you find this out?

EXERCISE 17.18

After having completed both Examples 17.3 and 17.4, what UNIX command(s) would enable you to update the repository **test** from Example 17.3 with what is in your online GitHub repository named **test**?

EXERCISE 17.19

What Git command can you use to see the abbreviated list of commits in the current branch of a repository and their commit comments?

Example 17.5: Pulling from a GitHub Repository

Objectives: To show the mechanics of taking content from a GitHub repository branch and adding it to a local repository by merging it with a local repository branch.

Introduction: The easiest way to share content from a GitHub repository is to use the `git pull` command. This command combines `git fetch` and `git merge` so that the content of a GitHub repository branch can be duplicated on a branch of one of your local repositories. We create a new local working directory and repository in it to receive the content from a remote source on GitHub. We then use the GitHub repository **https://github.com/bobk48/unixthetextbook3**, which contains all of the source code examples for the book you are now reading, as the remote source.

Git Commands Referenced: Table 17.5 shows the Git commands, and a brief description of each, that are used in this example. It is arranged in the order presented. Any argument enclosed in < > is a string of text. In order to get a more complete description of all the commands in the table, you can look at the man page for a particular command. For example, `man git-pull` gives you a complete man page for the `git pull` command.

Prerequisites: The following are the prerequisites for carrying out this example:

1. Knowing how to navigate to www.github.com using a Web browser GUI interface

TABLE 17.5 Git Commands Referenced

Command	Description
`git init`	Creates the **.git** directory in the working directory, initializing the data structures and objects necessary for a repository to exist
`git status`	Reports on the differences between files in the working directory and the index, and what files are untracked
`git add <file>`	Stages a file to the index
`git commit -m "<Message>"`	Takes a snapshot of the index, both files and directories, with `<Message>` automatically added
`git pull <ref> master`	Retrieves the branch named **master** from the remote `<ref>` designated into the current branch

2. Having completed the previous subsections that familiarize you with Git commands executed on the UNIX command line
3. Having an Internet connection and a suitable Web browser installed and operating on your UNIX system
4. Having completed Examples 17.3 and 17.4

Procedure: Carry out the following steps, in the order presented, to meet the objectives of this example:

1. Begin by setting up a new local repository working directory and initializing it as a Git repository.

```
% mkdir unixthetextbook3
% cd unixthetextbook3
% git init
Initialized empty Git repository in /usr/home/bob/
unixthetextbook3/.git/
%
```

2. Put a file in the new repository.

```
% touch Readme.txt
%
```

3. Examine the status of the new repository.

```
% git status
On branch master
Initial commit
Untracked files:
  (use "git add <file>..." to include in what will be
  committed)
        Readme.txt
nothing added to commit but untracked files present (use
"git add" to track)
%
```

4. Stage the **Readme.txt** file, and make your initial commit into the new repository.

```
% git add Readme.txt
% git commit -m "first commit"
[master (root-commit) 57e0400] first commit
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 Readme.txt
%
```

5. Use the `git pull` command to fetch and merge the entire **unixthetextbook3** repository from the branch named **master**.

```
% git pull https://github.com/bobk48/unixthetextbook3
master
From https://github.com/bobk48/unixthetestbook3
%
```

You are placed in the default editor. Leave the first line, numbered 1, as is, and save and quit the file.

```
  * branch master -> FETCH_HEAD
       1 Merge branch 'master' of https://github.com/
       bobk48/unixthetextbook3
       2
       3 # Please enter a commit message to explain why
       this merge is necessary,
       4 # especially if it merges an updated upstream into
       a topic branch.
       5 #
       6 # Lines starting with '#' will be ignored, and an
       empty message aborts
       7 # the commit.
/usr/home/bob/unixthetextbook3/.git/MERGE_MSG: 7 lines,
295 characters.
Merge made by the 'recursive' strategy.
 .gitattributes                                 |  22
 ++++++
 .gitignore                                     |  43
 ++++++++++++
 README.md                                      |   4 ++
 chap22_code/xlib_xcb_programs/1stxcbdraw.c    | 115
 ++++++++++++++++++++++++++++++
 chap22_code/xlib_xcb_programs/2ndxcb_events.c | 102
 +++++++++++++++++++++++++++
 chap22_code/xlib_xcb_programs/test1.c         |  59
 ++++++++++++++++
 chap22_code/xlib_xcb_programs/test2.c         |  21
 ++++++
 chap22_code/xlib_xcb_programs/test3.c         |  53
 ++++++++++++++
 chap22_code/xlib_xcb_programs/test4.c         |  99
 ++++++++++++++++++++++++++
 chap22_code/xlib_xcb_programs/test7.c         |  98
 ++++++++++++++++++++++++++
 chap22_code/xlib_xcb_programs/test8.c         | 132
 +++++++++++++++++++++++++++++++
 chap22_code/xlib_xcb_programs/xcb_events.c    | 146
 +++++++++++++++++++++++++++++
…
create mode 100644 qt/qt_progs3/qt_progs3
create mode 100644 qt/qt_progs3/qt_progs3.pro
create mode 100644 qt/qt_progs4/Makefile
create mode 100644 qt/qt_progs4/qt4.cpp
create mode 100644 qt/qt_progs4/qt4.o
create mode 100644 qt/qt_progs4/qt_progs4
create mode 100644 qt/qt_progs4/qt_progs4.pro
<Output Truncated>
```

6. Examine the contents of the working directory.

```
% ls
The current contents of the unixthetextbook3 repository
%
```

Conclusion: Using the `git pull` command, you can take content from a GitHub repository branch and put it on a local repository branch.

EXERCISE 17.20

What refspec URL and fetch assignments are listed for the repository **unixthetextbook3**? How did you find this out?

17.5.7.9 Solaris Git Installation Addendum

The following steps will allow you to install Git on a Solaris system:

1. Become the superuser.

```
bob@solaris:~$ su
Password: xxxxxx
```

2. Use the `pkg info` command to examine what Git is available in the default online repository for your Solaris. Note the name as **/developer/versioning/git**.

```
root@solaris:~# pkg info -r git
          Name: developer/versioning/git
       Summary: git - Fast Version Control System
   Description: Git is a free & open source, distributed
                version control system
                designed to handle everything from small to
                very large projects with speed and efficiency.
      Category: Development/Source Code Management
         State: Not installed
     Publisher: solaris
       Version: 1.7.9.2
 Build Release: 5.11
        Branch: 0.175.2.0.0.42.1
Packaging Date: June 23, 2014 01:29:36 AM
          Size: 22.91 MB
          FMRI: pkg://solaris/developer/versioning/
git@1.7.9.2,5.11-0.175.2.0.0.42.1:20140623T012936Z
root@solaris:~#
```

3. Use `pkg install` to install Git.

```
root@solaris:~# pkg install developer/versioning/git
           Packages to install:  1
```

```
              Create boot environment: No
       Create backup boot environment: No
                   Services to change:  1
DOWNLOAD              PKGS        FILES    XFER (MB)   SPEED
Completed              1/1      334/334      8.5/8.5  454k/s
PHASE                                          ITEMS
Installing new actions                       496/496
Updating package state database                 Done
Updating image state                            Done
Creating fast lookup database                   Done
root@solaris:~#
```

4. You now have Git on your Solaris system.

17.6 STATIC ANALYSIS TOOLS

Static analysis of a program involves analyzing the structure and properties of your program without executing it. These analyses are usually meant to determine the level of portability of your code for multiple platforms, the number of *lines of code* (LOC), the number of *function calls/points* (FPs) in your program, and the percentage of time taken by each function in the code. During the planning phase of a software project, parameters such as LOC and FPs are commonly used in software cost models that are used to estimate the number of person-months needed to complete a software project and, hence, the software cost.

Static analysis tools allow you to measure those parameters. In the following sections, we discuss some useful UNIX tools that allow you to perform these analyses. Our focus is the `lint` utility, but we also briefly describe the `prof` utility.

17.6.1 Verifying Code for Portability

Most C compilers do fairly well at checking for type mismatches, but few handle portability. You can use the `lint` utility to check your C software for portability. It is one of the most useful tools in UNIX for developing high-quality, clean, and portable C software, yet it is one of the least understood and used. It detects program features that are likely to be bugs, nonportable, or wasteful of system resources. Although `lint` can be used with many types of files, including C program, assembly, and preprocessor files, the discussion here is limited to its use with C program files. It is not available in Solaris. Thus, we show the use of this tool under PC-BSD.

In addition to performing tight type checking, `lint` also performs many other checks on a program to report structural problems such as unreachable statements, loops that are not entered at the top, local variables declared and not used, and logical expressions whose values are constant. The `lint` utility also reports messages if it finds functions that return values in some places and not in others, functions that are called with varying numbers or types of arguments, and functions that return values that are not used or whose return values are used but are not supposed to return any.

Most of the messages reported by `lint` are meaningful; they tell you what the problem is and where it is. However, some of its messages are difficult to understand and irrelevant.

You simply have to learn to ignore such messages. The following is a brief description of the `lint` utility.

SYNTAX

`lint [options] file-list`

Purpose: Allows checking of C programs, specified in **file-list**, for features that can be bugs, nonportable, or wasteful of system resources

Commonly used options/features:

- **`-c`** Check type casts (coercions) of questionable validity
- **`-s`** Produce one-line error messages (or warnings) only
- **`-u`** Suppress complaints about external variables and functions used and not defined, or vice versa; useful for running `lint` on a subset of modules of a software
- **`-v`** Suppress complaints about unused arguments in functions

We demonstrate the use of `lint` with the simple program shown in the following session and stored in the **cat.c** file. The program reads input from **stdin** and sends it to **stdout**. It is in a sense, then, a simple version of the `cat` command. We have used the `nl` command with `-ba` option to number all the source lines (including blank lines) because the line numbers reported in `lint's` error messages include blank lines.

```
% nl -ba cat.c
     1     /* Copy stdin to stdout */
     2
     3     #include <stdio.h>
     4
     5     int main(int argc, char *argv[])
     6     {
     7             char c;
     8             int    i, j;
     9
    10             while ((c=getchar()) != EOF)
    11                 putchar(c);
    12     }
%
```

The program compiles without any error messages from the compiler. It also runs without a problem, echoing each line entered from the keyboard until you press `<Ctrl+D>`, the EOF character in UNIX on a new line. The compilation and execution of the program is shown in the following session.

```
% a.out
Hello there!
Hello there!
Let's see how it goes.
```

```
Let's see how it goes.
That's all, folks!
That's all, folks!
<Ctrl+D>
%
```

We now use the `lint` command to see if it detects any potentially problematic features in the **cat.c** program. The following is a run of `lint` on **cat.c**.

```
% lint cat.c
cat.c:
cat.c(8): warning: i unused in function main [192]
cat.c(8): warning: j unused in function main [192]
cat.c(12): warning: function main falls off bottom without
returning value [217]
cat.c(5): warning: argument argc unused in function main [231]
cat.c(5): warning: argument argv unused in function main [231]
_types.h(61): warning: struct __timer never defined [233]
_types.h(62): warning: struct __mq never defined [233]
stdio.h(142): warning: struct pthread_mutex never defined [233]
stdio.h(143): warning: struct pthread never defined [233]
lint: cannot find llib-lc.ln
Lint pass2:
__srget used( cat.c(10) ), but not defined
__stdinp used( cat.c(10) ), but not defined
putc used( cat.c(11) ), but not defined
__swbuf used( cat.c(11) ), but not defined
getc used( cat.c(10) ), but not defined
__stdoutp used( cat.c(11) ), but not defined
__isthreaded used( cat.c(10) ), but not defined
%
```

The first five lines of the output are warnings about your program source. The warnings are pretty self-explanatory and may be removed by making the necessary changes in the source code. The warning at line 12 means that there is no explicit `return` statement for the `main()` function. The remaining esoteric warnings and comments are about the header files and library functions and may be ignored. In the following session, we show the new version of the program, the output of the `lint` command when it is used with the new version, compilation of the new code, and a sample run. Note that `lint` did not report a single error. This is how production-quality C code should be developed on UNIX platforms.

```
% nl -ba cat_new.c
     1      /* Copy stdin to stdout */
     2
     3      #include <stdio.h>
     4
     5     int main()
     6     {
     7            char c;
     8
     9            while ((c=getchar()) != EOF)
    10                (void) putchar(c);
    11
    12            return(0);
    13     }
% lint cat_new.c
cat_new.c:
_types.h(61): warning: struct __timer never defined [233]
_types.h(62): warning: struct __mq never defined [233]
stdio.h(142): warning: struct pthread_mutex never defined [233]
stdio.h(143): warning: struct pthread never defined [233]
lint: cannot find llib-lc.ln
Lint pass2:
__srget used( cat_new.c(9) ), but not defined
__stdinp used( cat_new.c(9) ), but not defined
putc used( cat_new.c(10) ), but not defined
__swbuf used( cat_new.c(10) ), but not defined
getc used( cat_new.c(9) ), but not defined
__stdoutp used( cat_new.c(10) ), but not defined
__isthreaded used( cat_new.c(9) ), but not defined
% cc cat_new.c
% a.out
Long live lint!
Long live lint!
%
```

The following in-chapter exercise is designed to give you an appreciation of the use of the `lint` utility and to help you understand some of the error messages that it produces.

EXERCISE 17.21

Go through all the sessions presented in this section to appreciate how `lint` works. Does `lint` produce the same error messages on your system for the first version of **cat.c**?

You can put some special comments in your code that are treated specially by `lint`. When `lint` reaches these special comments, it takes an action specific to the comment. We discuss one special comment that informs `lint` of functions that never return.

System calls such as `exit()` and `exec()` that do not return are not understood by `lint`, nor is the `return` call. This condition causes a different type of wrong error reports (or warnings) from `lint`. The `/*NOTREACHED*/` comment can be placed after such calls, informing `lint` that this path through the program code can never be reached.

When this comment is read by `lint`, it does not produce the bogus warning. Use of `/*NOTREACHED*/` is shown in the following session.

```
% cat sample.c
...
    if (fd == -1) {
        printf("File open failed.");
        exit(0);
        /*NOTREACHED*/
    }
    ...
%
```

The `lint` command can be run with several command line options. For example, the `-c` option checks type casts of questionable validity. Thus, `lint` reports a warning for the `s = (int *) i;` statement in the following code:

```
$ cat test.c
int main()
{
    char *s;
    int i=100;

    s = (int *) i;

    return(0);
}
% lint test.c
test.c:
test.c(6): warning: illegal pointer combination, op = [124]
test.c(6): warning: s set but not used in function main [191]
lint: cannot find llib-lc.ln
Lint pass2:
$ lint -c test.c
(8) warning: illegal combination of pointer and integer, op CAST
(9) warning: assignment type mismatch: pointer to char "=" pointer
to int
$
```

We strongly recommend that you create a make rule for running the `lint` utility on your modules before compiling them. The following is a sample make rule and its execution:

```
% cat makefile
SOURCES = compute.c input.c main.c
LINTFLAGS = -c
...
lint: $(SOURCES)
```

```
lint $(LINTFLAGS) $(SOURCES)
...
% make lint
lint -c compute.c input.c main.c
...
%
```

Although `lint` checks for most portability features, it does not check a few things. It does not check whether control strings in the `printf` calls match the types of the corresponding variables. Nor does it ensure that variables are unique after the first seven characters. Other than these minor exceptions, `lint` is trouble free and should be used regularly to remove sticky stuff from production software.

17.6.2 Source Code Metrics

You can use the UNIX tool `prof` to display a profile of your code in terms of the functions used and the percentage of time taken by each function. This information allows you to focus more closely on those functions that are causing bottlenecks in the software.

At times, you will want to know how long a program spends in each function when it is executed. You can use this information to improve the performance of certain portions of a program by optimizing them. UNIX has two main tools for analyzing the program performance: `prof` and `gprof`. Both tools enable your program to track down the number of times each function is called and the time spent in each function. The `gprof` tool provides more data than `prof`, but both are effective in identifying expensive portions of your program and execution paths in it. The use of both tools is very similar, and they generate similar output. We discuss `prof` only, but the steps shown work for `gprof` as well.

The first step in using `prof` is to compile your program with a particular option that asks the compiler to insert appropriate code in the object module for counting the number of times that each function is executed and the time spent in each function. For fully testing your program, use the `cc` compiler command with the `-p` option, as in:

```
% cc -p matrix_mult.c -o mm
%
```

After your program has compiled successfully, run it. Execution produces the run-time data in a file called **mon.out** in a format that `prof` can read. You then use the `prof` utility with this file to display the program profile. The `gprof` tool is available on both PC-BSD and Solaris. However, `prof` is available on Solaris only.

17.7 DYNAMIC ANALYSIS TOOLS

Dynamic analysis of a program involves its analysis during run time. As we mentioned before, this phase comprises debugging, tracing, and performance monitoring of the software, including testing it against product requirements. In this section, we discuss the two useful UNIX tools for tracing the execution of a program and debugging (`gdb`), and measuring the running time of a program in actual time units (`time`).

17.7.1 Source Code Debugging

The task of debugging software is time-consuming and difficult. It consists of monitoring the internal working of your code while it executes, examining values of program variables and values returned by functions, and executing functions with specific input parameters. As we stated before, many C programmers tend to use the `printf` calls (`cout` for C++) at various places in their programs to locate the origin of a bug and then remove it. This technique is simple and works quite well for small programs. However, for large-size software, where an error may be hidden deep in a function call hierarchy, this technique ends up taking a lot of editing time for adding and removing `printf` (or `cout`) calls in the source file. A more efficient debugging method under such circumstances is to use a symbolic debugger. A typical *symbolic debugger* offers several facilities for observing the run-time behavior of a program, including the following:

- Running programs
- Setting breakpoints
- Single stepping
- Listing source code
- Editing source code
- Accessing and modifying variables
- Tracing program execution
- Searching for functions and variables
- Identifying what a program was doing when it crashed

Several symbolic debuggers are available on UNIX platforms, the most common being the freeware GNU debugger, gdb. The default source code debuggers on Solaris and PC-BSD are adb and gdb, respectively. They offer similar facilities. adb is a link to mdb, the *modular debugger*, which allows you to examine processes, user process core dumps, as well as live operating system and operating system dumps. We primarily describe gdb, as it is the standard debugger on most UNIX systems. Although gdb has several features for debugging C++ classes as well, we discuss its features for debugging C programs only. The following is a brief description of the utility.

SYNTAX

```
gdb [options] [executable-file [core-file or PID]]
```

Purpose: Allows source-level debugging and execution of the program in **executable-file**, which was generated using a C or C++ source file; or **core-file**, created due a C/C++ program crash; or process ID (**PID**) of a running program

Commonly used options/features:

-c file	Examine the file **file** as the core dump (i.e., file created by UNIX when a program crashes)
-h	List all options along with their brief explanations
-n	Do not run commands from any **.gdbinit** initialization files
-x file	Execute gdb commands from the file **file** (or from the **.gdbinit** initialization file(s) if **file** is not specified)

During startup, gdb searches for **.gdbinit**. The search order is: your current directory (**.**) and then your home directory (~). The -x option allows you to use any file as a startup file. Table 17.6 gives a brief description of some of the commonly used gdb commands.

17.7.1.1 Using gdb

Before debugging a program with gdb (or any other debugger), you must compile it with the -g compiler option to include the symbol table and relocation, debugging, and profiling information in the executable. This information is used by the debugging and profiling tools. We use the program in the **bugged.c** file to show various features of gdb. The program prompts you for keyboard input, displays the input, and exits. We use several functions to demonstrate the features of gdb for setting breakpoints and displaying the stack trace at function boundaries, tracing program execution by executing program statements one by one, viewing types and values of variable, and so on. The following session shows the program code, its compilation without the -g option, and its execution.

```
% nl -ba bugged.c
     1      /*
     2       *  Sample C program bugged with a nasty error
     3       */
     4
     5
     6      #include <stdio.h>
     7
     8      #define PROMPT     "Enter a string: "
     9
    10      void get_input(char *, char *);
    11      void null_function1 ();
    12      void null_function2 ();
    13
    14      int main ()
    15      {
    16              char *s_val, *temp;
    17
    18              temp = s_val;
    19              null_function1 ();
```

TABLE 17.6 Commonly Used gdb Commands

Command	Brief Description		
break	break <line_num>	:	Set breakpoint at line number line_num
	break <function_name>	:	Set breakpoint at function function_name
continue	continue	:	Continue execution after breakpoint
clear	clear <line>	:	Delete breakpoint set at line number line
	clear <function>	:	Delete breakpoint set at function function
delete	delete <num>	:	Delete breakpoint number num
	delete	:	Delete all breakpoint numbers
frame	frame	:	Show all stack frames
	frame <num>	:	Set current stack frame to frame number num
help	help <num>	:	List a brief description of the command classes
	help command	:	Display a brief description of a command or command class command
info	info break	:	Show information about current breakpoints
	info frame	:	Show information about current stack frame
	info locals	:	Show contents of local variables on the current stack frame
	info registers	:	Display values of CPU registers
list	list	:	List next few (10 by default) lines of the program
	list <line>	:	List 10 lines around line number line
	list <start>,<end>	:	List 10 lines from lines start through end
	list <function>	:	List 10 lines of the function
next	next	:	Like the step command, except that it treats a function call as one instruction
	next <count>	:	
print	print <expr>	:	Display the value of the expression expr
	print identifier	:	Display the current value of identifier
quit	quit	:	Quit gdb
set	set <var> = <expr>	:	Set variable var to expression expr
step	step	:	Execute the next program instruction, stepping into a function (i.e., not treating a function call as one instruction)
	step <count>	:	Execute next count lines of program code
run	run [command-line-args]	:	Execute the program that was an argument of the gdb command
whatis	whatis identifier	:	Display the type of identifier

```
        null_function2 ();
        get_input(PROMPT, temp);
        (void) printf("You entered: %s\n", s_val);
        (void) printf("The end of buggy code!\n");
        return (0);
}

void get_input(char *prompt, char *str)
{
        (void) printf("%s", prompt);
        for (*str = getchar(); *str != '\n'; *str =
              getchar())
              str++;
        *str = '\0'; /* string terminator */
}

void null_function1 ()
{ }

void null_function2 ()
{ }

% cc bugged.c -o bugged
% bugged
Enter a string: Need gdb!
Segmentation Fault
%
```

Note that the program prompts you for input and faults without echoing what you enter from the keyboard. This happens frequently in C programming, particularly with programmers who are new to C or are not careful about initializing variables in their programs and rely on the compiler. It is time to use gdb!

17.7.1.2 Entering the gdb Environment

As we stated earlier, in order to enter the gdb environment, you must compile your C program with the `-g` compiler option. This option creates an executable file that contains the symbol table and debugging, relocation, and profiling information for your program. After the source program compiles successfully, you can then use the `gdb` command to debug your code, as in the following session. We ran the `gdb –q` command to prevent the introductory messages from being displayed. Note that `(gdb)` is the prompt for the gdb debugger.

```
% cc -g bugged.c -o bugged
% gdb -q bugged
(gdb)
```

Once inside the gdb environment, you can run many commands to monitor the execution of your code. You can use the `help` command to get information about the gdb commands. Without any argument, the `help` command displays the names of all of the gdb *command classes*. A command class specifies the type of operations you can perform on your executable code, a core file, or a process. Under a command class, you can run the commands available for that class. You can get information about any gdb command class by passing the command name as an argument to the `help` command. In the following session, the `help` command shows the names of all gdb commands and the `help tracepoints` command displays a brief description of the command class `tracepoints` and commands supported by gdb under this class.

```
(gdb) help
List of classes of commands:
aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the
program
user-defined -- User-defined commands
Type "help" followed by a class name for a list of commands in
that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb) help tracepoints
Tracing of program execution without stopping the program.
List of commands:
actions -- Specify the actions to be taken at a tracepoint
collect -- Specify one or more data items to be collected at a
tracepoint
delete tracepoints -- Delete specified tracepoints
disable tracepoints -- Disable specified tracepoints
enable tracepoints -- Enable specified tracepoints
end -- Ends a list of commands or actions
passcount -- Set the passcount for a tracepoint
save-tracepoints -- Save current tracepoint definitions as a
script
tdump -- Print everything collected at the current tracepoint
tfind -- Select a trace frame;
tfind end -- Synonym for 'none'
tfind line -- Select a trace frame by source line
tfind none -- De-select any trace frame and resume 'live'
debugging
tfind outside -- Select a trace frame whose PC is outside the
given range
tfind pc -- Select a trace frame by PC
tfind range -- Select a trace frame whose PC is in the given range
tfind start -- Select the first trace frame in the trace buffer
tfind tracepoint -- Select a trace frame by tracepoint number
trace -- Set a tracepoint at a specified line or function or
address
tstart -- Start trace data collection
tstatus -- Display the status of the current trace data collection
tstop -- Stop trace data collection
while-stepping -- Specify single-stepping behavior at a tracepoint
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb) help trace
Set a tracepoint at a specified line or function or address.
Argument may be a line number, function name, or '*' plus an
address.
For a line number or function, trace at the start of its code.
If an address is specified, trace at that exact address.
Do "help tracepoints" for info on other tracepoint commands.
(gdb)
```

In addition to the gdb-specific commands, gdb also allows you to execute all shell commands.

17.7.1.3 Executing a Program

You can run your program inside the gdb environment by using the `run` command. The following command executes the program called **bugged**.

```
(gdb) run
Starting program: /usr/home/sarwar/unix3e/ch17/Gdb/bugged
Enter a string: Need gdb!
Program received signal SIGSEGV, Segmentation fault.
0x0000000000400951 in get_input (prompt=0x400a76 "Enter a string:",
str=0x0) at bugged.c:30
30     for (*str = getchar(); *str != '\n'; *str = getchar())
Current language: auto; currently minimal
(gdb)
```

The program (now a process) prompts you for input. When you enter the input (`Need gdb!` in this case) and hit `<Enter>`, the process fails when it tries to execute the command at line 30. The error message is quite cryptic for beginners and those who are not familiar with UNIX jargon. All the error message says is that a signal of type `SIGSEGV` (segmentation violation, i.e., address space violation) was received by the process when it was executing the statement at line 30. The message explains that `SIGSEGV` was sent to the program because of `Segmentation fault`. A segmentation fault is generated by the UNIX kernel when a process tries to access a memory region (a segment) that it is not allowed to access. In other words, your process tried to access a memory location that did not belong to its *address space*.

17.7.1.4 Listing Program Code

You can use the `list` command to display all or part of a source program. By default, it displays 10 lines around the line (or function) that you specify as its argument. You can display lines of code in a particular function or on a range of lines. In the following example, the `list get _ input` command is used to display 10 lines in the code around the `get _ input` function, and the `list 14,27` command is used to display the source program at lines 14–27. Note that the `list get _ input` command displays five lines before the start of the function code and five lines after it. In order to display the first 10 lines of the function code, we use the `list get _ function,` command. Since `get _ function` has less than 10 lines, a few extra lines after the function are also displayed. Use the `help list` command to get more information about the `list` command.

```
(gdb) list get_input
23          (void) printf("The end of buggy code!\n");
24          return (0);
25   }
26
27   void get_input(char *prompt, char *str)
28   {
29          (void) printf("%s", prompt);
30          for (*str = getchar(); *str != '\n'; *str = getchar())
31                 str++;
32          *str = '\0'; /* string terminator */
(gdb) list get_input,
28   {
29          (void) printf("%s", prompt);
30          for (*str = getchar(); *str != '\n'; *str = getchar())
31                 str++;
32          *str = '\0'; /* string terminator */
33   }
34
35   void null_function1 ()
36   { }
37
(gdb) list 14,27
14   int main ()
15   {
16          char *s_val, *temp;
17
18          temp = s_val;
19          null_function1 ();
20          null_function2 ();
21          get_input(PROMPT, temp);
22          (void) printf("You entered: %s.\n", s_val);
23          (void) printf("The end of buggy code!\n");
24          return (0);
25   }
26
27   void get_input(char *prompt, char *str)
(gdb)
```

17.7.1.5 Tracing Program Execution

To find out what went wrong in our process, we need to identify the part of the code that may be problematic. There are several ways of doing so, including line-by-line tracing of the whole program, tracing statements of a function, calls to a particular function, and changes to a variable. We do so by backtracking the program using the `where` or `backtrace` command of gdb, as shown in the following session. The output of both commands is the same: the location of the program where the fault occurred and how this location was reached. The how part is identified by the function call sequence identified in the *stack trace* (also known as *stack frame* or *activation record* in the programming language jargon) that the command displays. Line #0 shows the top of stack with the `get _ input()` function's statement where the crash occurred, and #1 shows that the `get _ input()` function was called at line 21 in the `main()` function. The `main()` function is called the *caller* and the `get _ input()` function is called the *callee*.

```
(gdb) where
#0 0x0000000000400951 in get_input (prompt=0x400a76 "Enter a
   string: ",     str=0x0) at bugged.c:30
#1 0x0000000000400862 in main () at bugged.c:21
(gdb) backtrace
#0 0x0000000000400951 in get_input (prompt=0x400a76 "Enter a
   string: ", str=0x0) at bugged.c:30
#1 0x0000000000400862 in main () at bugged.c:21
(gdb)
```

The output of the command shows that the program was at location (memory address) 0x0000000000400951 at line 30 in the `get _ input()` function when it received the `SIGSEGV` signal. The stack frame shows that the call sequence is `main() => get _ input()` and the program was executing the `*str = getchar()` statement at line 30 when it crashed (see the program listing in the previous section). Note that the code on this line has two assignment statements and one comparison statement. In all of these statements, we dereference the string pointer `str`. Two statements use the C library function `getchar()`. This library function is well tested and has been in use for many years. Thus, the problem must be with the pointer variable `str`. We pursue this issue in a later section.

17.7.1.6 Setting Breakpoints

Viewing execution of all or part of your code statement by statement is called *program/code tracing*. In order to trace code, you need to set breakpoints in your program. You can trace a program up to a particular statement or function by using the `break` command, as described in Table 17.1. It allows you to run a program without interruption until the control reaches the line or function that you want to study more closely. The process of stopping a program in this way is known as setting *breakpoints*. As shown in the previous section, the main function starts at line 14 but its first executable statement is at line 18. We set the breakpoint at the first executable statement in `main()` and run the program. The program stops execution at line 18, the only breakpoint we have set, having statement `temp = s _ val;`.

```
(gdb) break main
Breakpoint 1 at 0x40083f: file bugged.c, line 18.
(gdb) run
Starting program: /usr/home/sarwar/unix3e/ch17/Gdb/bugged

Breakpoint 1, main () at bugged.c:18
18          temp = s_val;
Current language:  auto; currently minimal
(gdb)
```

17.7.1.7 Single-Stepping through Your Program

Always set breakpoints in your program to be able to view execution of all or part of your code statement by statement. The process of tracing program execution statement by statement is known as *single-stepping* through your program. Single-stepping, combined with tracing of variables, allows you to study program execution closely. Single-stepping can be done with the `step` command, which executes the next program statement, stepping into a function if the statement is a function call. You can use the `next` command to single-step through your code, but it executes a function into its entirety. If you are tracing a variable, it shows you the value of the variable when a statement within the scope of the variable executes. Run the `help  scope` command to get more information about the scoping rules.

After setting the breakpoint at `main` and running the program in the previous session, we then run each statement of the program one by one by using the `next` command. After the third `next` statement, the control reaches the call to the `get _ input  (PROMPT, temp);` function at line 21 in the `main()` function. We then use the `step` command to step into and execute each statement in the `get _ input()` function. The first `step` command takes control to the first executable statement in the `get _ input()` function, the `for` loop. The second `step` statement prompts you for keyboard input. As soon as you hit `<Enter>` after typing `Hello  World!`, the system displays the error message `Program received signal SIGSEGV, Segmentation fault.` along with the problematic source code statement and line number (30 in this case) and the value of the `str` variable. The hexadecimal number at the beginning of the second line of the error message is the memory address (**0x0000000000400951**) of the `getchar()` function that caused the exception. We can use the `x` command to display the contents of this memory location. The output of the command shows that the error occurred at the 178th byte in the `getchar()` function. We still do not know the reason why the program faulted.

```
(gdb) next
19          null_function1 ();
(gdb) next
20          null_function2 ();
(gdb) next
21          get_input(PROMPT, temp);
(gdb) step
get_input (prompt=0x400a76 "Enter a string: ", str=0x0) at
bugged.c:29
29          (void) printf("%s", prompt);
(gdb) step
30          for (*str = getchar(); *str != '\n'; *str = getchar())
(gdb) step
Enter a string: Hello world!

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400951 in get_input (prompt=0x400a76 "Enter a string:
", str=0x0) at bugged.c:30
30          for (*str = getchar(); *str != '\n'; *str = getchar())
(gdb) x 0x0000000000400951
0x400951 <get_input+177>:     0x8b480a88
(gdb)
```

17.7.1.8 Accessing Identifiers (Variables and Functions)

You can access the location of an *identifier* (variable or function) in the program source and view its type, value, and places of use by using various gdb commands. In the following session, we illustrate the use of these commands with examples. The outputs of the commands are fairly self-explanatory. The `print s _ val` and `print str` commands display the values of the `s _ vala` and `str` variables as zero each and the `whatis str` command displays the type declaration of the `str` variable: `char *`, as expected.

```
(gdb) delete
Delete all breakpoints? (y or n) y
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /usr/home/sarwar/unix3e/ch17/Gdb/bugged
Enter a string: Hello world!

Program received signal SIGSEGV, Segmentation fault.
```

```
0x0000000000400951 in get_input (prompt=0x400a76 "Enter a string:
", str=0x0) at bugged.c:30
30          for (*str = getchar(); *str != '\n'; *str = getchar())
(gdb) whatis str
type = char *
(gdb) print s_val
$1 = 0x0
(gdb) print str
$1 = 0x0
(gdb)
```

17.7.1.9 Fixing the Bug

After finding out that the program faults at line 30, the first thing you should do is determine the variables involved in the statement that caused the fault. Then you should display the values of theses variables.

The session immediately reveals that the value of the actual parameter to the get _ input() function, s _ val, is nil (0). This causes the formal parameter in the get _ input() function, str, to have a starting value of nil. When we dereference the str variable to store user input, we try to access memory location with address **0**. This location belongs to the UNIX kernel space and is used to store the resident part of the operating system. Therefore, the process tries to write to a location that is outside its address space—that is, does not belong to it. This attempt is a clear violation that results in the SIGSEGV signal sent to the running program, causing its termination—the default action on this signal. Hence, you see the error message Segmentation Fault when you run the program from the command line. Figure 17.15 illustrates segmentation violation.

To fix the bug in the program, all you need do is initialize the s _ val pointer to a memory space that has been allocated to the program, statically or dynamically. We use a character array called user _ input[SIZE] and set the s _ val pointer to point to

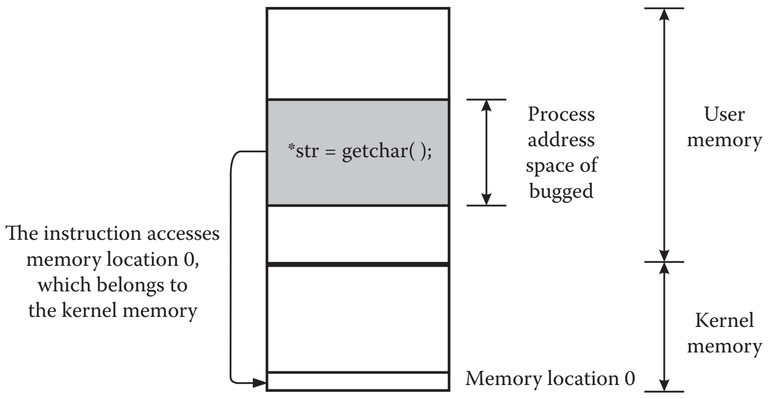

*str = getchar( );
Process address space of bugged
User memory
Kernel memory
Memory location 0
The instruction accesses memory location 0, which belongs to the kernel memory

FIGURE 17.15 The memory (segmentation) access violation causing program failure.

the first byte of the array. The revised `main()` function is shown in the following session, along with its compilation and proper execution. The changes in the code are the additions of lines 8, 17, and 19 in the program, as follows.

```
% nl -ba working.c
     1       /*
     2        *  Sample C porgram bugged with a nasty error
     3        */
     4
     5       #include <stdio.h>
     6
     7       #define PROMPT "Enter a string: "
     8       #define SIZE   255
     9
    10       void get_input(char *, char *);
    11       void null_function1 ();
    12       void null_function2 ();
    13
    14       int main ()
    15       {
    16              char *s_val, *temp;
    17              char user_input[SIZE];
    18
    19              s_val = user_input; /* Initialize s_val to an
                    array */
    20              temp = s_val;
    21              null_function1 ();
    22              null_function2 ();
    23              get_input(PROMPT, temp);
    24              (void) printf("You entered: %s\n", s_val);
    25              (void) printf("The end of buggy code!\n");
    26              return (0);
    27       }
    28
    29       void get_input(char *prompt, char *str)
    30       {
    31              (void) printf("%s", prompt);
    32              for (*str = getchar(); *str != '\n'; *str =
                          getchar())
    33                   str++;
    34              *str = '\0'; /* string terminator */
    35       }
    36
    37       void null_function1 ()
    38       { }
    39
    40       void null_function2 ()
    41      { }
    42
% cc bugged.c -o working
% working
Enter a string: Hello World!
You entered: Hello World!
The end of buggy code!
%
```

17.7.1.10 Leaving gdb and Wrapping Up

You can use the `quit` command to leave gdb and return to your shell.

```
(gdb) quit
%
```

Once your code has been debugged, you can decrease the size of the binary file, releasing some disk space, by removing from it the information generated by the `-g` option of the C compiler to be used by debugging and profiling utilities. You can do so by using the `strip` command. The information stripped from the file contains the symbol table and relocation, debugging, and profiling information. In the following session, we show the long list for the working file before and after the execution of the `strip` command. Note that the size of the file has decreased from 8318 bytes to 5552 bytes, resulting in a saving of about 33% disk space. Alternatively, you can recompile the source to generate an optimized executable by using various options.

```
% ls -l working
-rwxr-xr-x 1 sarwar faculty 8318 Nov 10 08:25 working
% strip working
% ls -l working
-rwxr-xr-x 1 sarwar faculty 5552 Nov 10 08:27 working
%
```

In the following in-chapter exercise, you will make extensive use of gdb to understand its various features.

EXERCISE 17.22

Go through all gdb commands discussed in this section to appreciate how gdb works. If some of the commands used in this section do not work on your system, use the `help` command to list the gdb commands and use those that are available in your version of gdb.

17.7.2 Run-Time Performance

The run-time performance of a program or any shell command can be measured and displayed by using the `time` command. This command reports three times: *real time*, *system time*, and *user time* in the format hours:minutes:seconds. Real time is the actual time taken by the program to finish running, system time is the time taken by system (kernel) activities while the program was executing, such as handling the clock interrupt, and user time is the time taken by execution of the program code. Because UNIX is a time-sharing system, real time is not always equal to the sum of system and user time, as many other user processes may be running while your program executes. The following is a brief description of the command.

SYNTAX

```
time [command]
```

Purpose: Report the run-time performance of `command` in terms of its execution time. It reports three times: real time (actual time taken by command execution), system time (time spent on system activities while the command was executing), and user time (time taken by the command code itself).

There are two versions of the `time` command: the built-in command for the C shell and the `/usr/bin/time` command. The output of the built-in `time` command is quite cryptic, whereas the output of the `/usr/bin/time` command is very readable. When the C shell version of the `time` command is executed without a command argument, it reports the length of time the current C shell has been running. The reported time includes the time taken by all its children—that is, all the commands that have run under the shell. The other version of the command does not have this feature. The `time` command sends its output to **stderr**. So, if you want to redirect the output of the `time` command to a disk file, you must redirect its **stderr** (not its **stdout**) to the file.

The following `time` command, executed under the C shell, reports how long the current shell has been running: 36 minutes and 53.51 seconds. In the output, `u` represents *user* time and `s` represents *system* time.

```
% time
0.804u 0.540s 36:53.51 0.0%   5812+327k 9+122io 15pf+0w
%
```

The following command reports the time taken by the `find` command. For the sake of brevity, we have not displayed the error messages generated by the `find` command because of improper access privileges for certain directories. Note how neat the output looks.

```
% ls -l bigdata
-rw-r--r--  1 sarwar  faculty  1678770176 Nov 10 08:37 bigdata
% /usr/bin/time cp bigdata bigdata.old
       20.94 real         0.00 user         2.55 sys
%
```

As stated earlier, the sum of the user and system times does not always equal real time, especially if a program is idle and does not use the CPU for some time. In the output of the first time command, the real time is 20.94 seconds, which clearly does not equal the sum of the user and system times (0.00 and 2.55 seconds, respectively).

Because the time command can be used to measure the running time of any program, you can use it with an executable of your own—a binary image or a shell script. The following session shows the running time of the **quick_sort** program when it is executed to sort numbers in the **in_data** file. Note that the real time equals the system time plus the user time, as the command was run late at night when the system was not running any other user processes. In the second command, we copy:

```
% /usr/bin/time quick_sort in_data
       51.2 real         48.6 user         2.6 sys
%
```

There are other ways of measuring the running time of a program that give you better precision. But, using the time command to perform this task is the easiest way, and we certainly recommend it for beginners.

17.8 WEB RESOURCES

Table 17.7 lists useful websites for various programming languages and UNIX commands and tools for program development.

SUMMARY

UNIX supports all contemporary high-level languages (both interpreted and compiled), including C, C++, Java, Javascript, FORTRAN, BASIC, and LISP. We described the translation process that a program in a compiled language such as C has to go through before it can be executed. We also described briefly a typical software engineering life cycle and discussed in detail the program development process and the tools available in UNIX for this phase of the life cycle. The discussion of tools focused on their use for developing production-quality C software.

The program development process comprises three phases: code generation, static analysis, and dynamic analysis. The UNIX code generation tools include text editors (emacs, pico, and vi), C language enhancers (cb and indent), compilers (cc, gcc, xlc, CC, cpp, and g++), tools for handling module-based software (make), tools for creating libraries (ar, nm, and ranlib), and the most commonly used version control tool, Git/Github, and its related commands.

Git is a source code maintenance tool. It is a software database for tracking changes made to a set of source code files over time. Although it is most often used by programmers to coordinate changes to software source code, you can use Git to track any kind of content. Git perform the following tasks:

- Examines the state of your source code project at earlier points in time
- Shows the differences among various states of the project and the files present at those states

TABLE 17.7 Web Resources for the Most Commonly Used Programming Languages, and UNIX Commands and Tools for Program Development

General Pages: Shell Commands, Shell Programming, Advanced UNIX, and Program Development Tools	
www.tutorialspoint.com/unix/unix-manpage-help.htm	A Web page for UNIX shell commands, UNIX shell programming, advance UNIX, and many important UNIX tools and resources
tacpa.org/notes/linux/devTools.html	A Web page for UNIX/Linux development tools
Programming Languages	
www.perl.com	The Perl language homepage
http://www.bell-labs.com/usr/dmr/www/chist.html	History of the C language by Dennis Ritchie
www.stroustrup.com/C++.html	C++ language page, maintained by Bjarne Stroustrup, designer of the language
www.oracle.com/java/index.html	The home page for Java
www.oracle.com/technetwork/java/index.html	Essentials of the Java programming language
www.gnu.org/software/java/java.html	GNU and the Java language
www.haskell.org/haskellwiki/Haskell	The Haskell programming language
www.eiffel.com	The Eiffel programming language home page
www.smalltalk.org	The Smalltalk programming language home page
www.visualbasic.org	Association of Visual BASIC Professionals page
gcc.gnu.org/fortran	Home page for GNU FORTRAN
www.sigapl.org	ACM SIGPLAN chapter on APL
en.wikipedia.org/wiki/AWK	Wikipedia page for AWK
www.grymoire.com/Unix/Awk.html	A nice tutorial on AWK
www.gnu.org/software/gawk	Page for GNU AWK (gawk)
www.tcl.tk	A useful page for TCL developers
HTML Activities	
www.w3.org/MarkUp	XHxTML2 working group home page
www.w3schools.com/html/	A comprehensive HTML5 tutorial
www.w3schools.com/html/html_intro.asp	Introduction to HTML
Compilers	
gcc.gnu.org/	The home page for gcc
gcc.gnu.org/java/	The home page for GNU Java compiler gcj
www.w3schools.com/html/	A comprehensive HTML5 tutorial
www.cprogramming.com/g++.html	Home pages for g++ compiler use
courses.cs.washington.edu/courses/cse373/99au/unix/g++.html	
docs.oracle.com/javase/7/docs/technotes/tools/windows/javac.html	Page for Java language compiler javac
Make and Library Tools	
www.tutorialspoint.com/makefile/index.htm	Nice tutorials on makefiles
www.cs.colby.edu/maxwell/courses/tutorials/maketutor/	

(*Continued*)

TABLE 17.7 (CONTINUED) Web Resources for the Most Commonly Used Programming Languages, and UNIX Commands and Tools for Program Development

General Pages: Shell Commands, Shell Programming, Advanced UNIX, and Program Development Tools	
`capone.mtsu.edu/csdept/FacilitiesAndResources/make.htm`	A small tutorial on the `make` utility
`heather.cs.ucdavis.edu/~matloff/UnixAndC/CLanguage/Make.html`	Very good tutorial on makefiles and libraries
gdb and GNU Tools for Software Testing	
`http://www.gnu.org/software/gdb/gdb.html`	The home page for gdb, the GNU project debugger
`http://www.gnu.org/manual/`	Online documentation for GNU packages
Revision Control with Git/GitHub	
`http://gitref.org/`	A comprehensive reference site for Git
`http://git-scm.com/`	Open source site for Git hosted on GitHub
`https://github.com/`	GitHub hosting site
`en.wikipedia.org/wiki/GitHub`	The Wikipedia site for GitHub

- Splits the project development into multiple independent lines, called branches, which can evolve separately
- Regularly recombines branch content by merging, or reconciling, the content changes made in two or more branches
- Allows many people to work on a project simultaneously, sharing and combining their work as needed, at any convenient time

Git and GitHub are atomic-level, distributed, content-oriented version control systems. *Atomic level* means that when you take a snapshot of the software, everything in it is captured in the snapshot at a single instance in time. *Distributed* means that the entire software package you are working with is available to all collaborators locally at all times. *Content oriented* means that when you join different branches of work on the software, only the content of lines in the files along the branches you are merging are considered. Git indicates merged-content conflicts with several useful strategies and mechanisms, and resolves them using several tools that are add-ons to Git. But only the people writing, building, and testing how the software works, and those people managing that process, are responsible for resolving merged-context conflicts.

The purpose of the static analysis phase is to identify features of the software that might be bugs or nonportable, and to measure metrics such as lines of code (LOC), function points (FPs), and repetition count for functions. The UNIX tools that can be used for this purpose include `lint`, `prof`, and `dprof`.

The purpose of the dynamic analysis phase is to analyze programs during their execution. The tools used during this phase are meant to trace program execution in order to debug them and measure their run-time performance in terms of their execution time. The commonly used UNIX tools for this phase of the software life cycle are debuggers (gdb, etc.) and tools for measuring running times of programs (`time`).

UNIX has several tools for other phases of a software life cycle, but a discussion of them is outside the scope of this textbook.

QUESTIONS AND PROBLEMS

1. What are the differences between compiled and interpreted languages? Give three examples of each.
2. Give one application area each for assembly and high-level languages.
3. Write the steps that a compiler performs on a source program in order to produce an executable file. State the purpose of each step. Be precise.
4. What are the `-o`, `-l`, and `-xO` options of the `cc` command used for? Give an example command line for each and describe what it does.
5. Give the compiler commands to create an executable called **prog** from C source files **myprog.c** and **misc.c**. Assume that **misc.c** uses some functions in the math library. What is the purpose of each command?
6. What are the three steps of the program development process? What are the main tasks performed at each step? Write the names of UNIX tools that can be used for these tasks.
7. Give a shell command that can be used to determine the LOC in the program stored in the **scheduler.c** file.
8. Write advantages and disadvantages of automating the recompilation and relinking process by using the `make` utility, as opposed to manually doing this task.
9. Consider the following makefile and answer the questions that follow.

```
CC = cc
OPTIONS = -xO4 -o
OBJECTS = main.o stack.o misc.o
SOURCES = main.c stack.c misc.c
HEADERS = main.h stack.h misc.h
polish: main.c $(OBJECTS)
 $(CC) $(OPTIONS) power $(OBJECTS) -lm
main.o: main.c main.h misc.h
stack.o: stack.c stack.h misc.h misc.o: misc.c misc.h
```

List the following:

a. Names of macros
b. Names of targets
c. Files that each target is dependent on
d. Commands for constructing the targets named in part (b)

10. For the makefile in Problem 9, give the dependency tree for the software.
11. What commands are executed for the **main.o**, **stack.o**, and **misc.o** targets? How do you know?
12. For the makefile in Problem 9, what happens if you run the `make` command on your system? Show the output of the command.
13. Use the `nm` command to determine the size of the `select` call in the socket library (**/usr/lib/libsocket.a**). What is the size of the code for the call?
14. Give the command line for determining the library that contains the function `strcmp`. What is the size of this function?
15. What would happen if you tried to check out the same file again using Git/GitHub? Why?
16. Create a three-branch repository of commits exactly like Figure 17.16. Use any number of text files that you modify between commits on the three branches. Keep the default name for the branch **master**, but name the other two branches **test** and **dev**, as seen in Figure 17.16.
17. Create a three-branch repository of commits exactly like Figure 17.17. Use any number of text files that you modify between commits on the three branches. Keep the default name for the branch master, but name the other two branches **test** and **dev**, as seen in Figure 17.17.

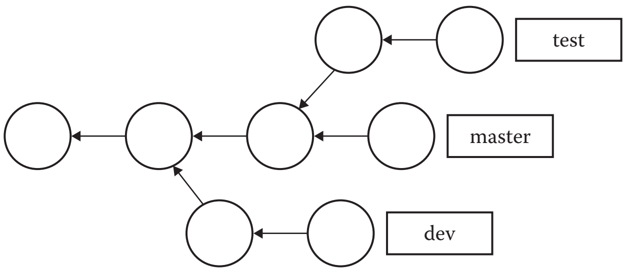


FIGURE 17.16 Sample three-branch repository of commits.

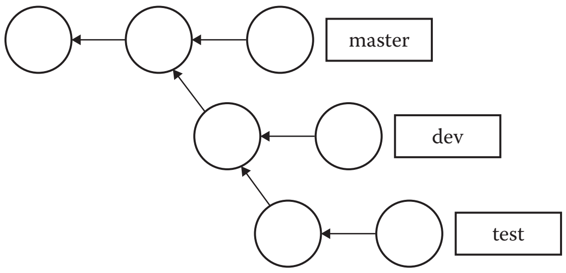


FIGURE 17.17 Sample three-branch repository of commits.

18. As an alternative to using `git pull` in Example 17.5 to obtain the source code for this book from the listed GitHub repository (**https://github.com/bobk48/unixthetextbook3**), use the `git clone` command, as shown in Example 17.4, from your home directory on your UNIX system. What will be the name of the repository directory created by Git on your local machine that contains the source code files?

19. Before doing this problem, go through all of the material in Chapter 24 on the Zettabyte File System, including the in-chapter exercises and problems. Then, in your own words, describe the differences and similarities between Git, GitHub, and ZFS. For example, the commands `zfs snapshot` and `git commit` are very similar. In what way do they differ? In your opinion, is it possible to use the commands `zpool` and `zfs` to achieve the same or similar results as using Git and GitHub? What other UNIX commands would be needed to augment the ZFS file system commands to attain results that are similar to Git and GitHub functions, commands, and capabilities?

20. The following code is meant to prompt you for integer input from the keyboard, read your input, and display it on the screen. The program compiles with one warning, but it doesn't work properly. Use the gdb utility to find the bugs in the program. What are they? Fix the bugs, recompile the program, and execute it to be sure that the corrected version works. Show the working version of the program.

```
#include <stdio.h>

#define PROMPT  "Enter an integer: " void get_input(char *,
int *);
void main ()
{
    int user_input;
    get_input(PROMPT, user_input);
   (void) printf("You entered: %d.\n", user_input);
}

void get_input(char *prompt, int *ival)
{
    (void) printf("%s", prompt);

    scanf ("%d", &ival);
}
```

21. What does the `time sh` command line do when executed under the C shell?

22. Give the command line to redirect the output of the `/usr/bin/time polish` command to a file called **polish_output**. Assume that you are using the Bourne shell.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

CHAPTER 18

System Programming I

File System Management

Objectives

- To explain the concept of system programming
- To describe briefly the concept of system calls
- To discuss the execution details of a system call in UNIX
- To describe briefly the types of system calls in UNIX
- To explain the concept of file and file descriptors in UNIX
- To discuss the concept of per-process file descriptor tables, system-wide file tables, and inode tables
- To discuss briefly standard I/O and low-level I/O
- To describe in detail the system calls for I/O of file data and attributes
- To describe briefly the purpose of system calls related to directories and setting file attributes
- To discuss the concept of file holes in UNIX
- To discuss several small programs to illustrate the use of the various system calls for the I/O and management of file data and attributes
- To discuss the concept of blocking I/O and restarting a system call
- To explain briefly several system calls and C library functions for manipulating directories and file attributes
- To cover the system calls, library calls, and primitives

```
access(), chdir(), chmod(), chown(), close(), closedir(),
creat(), fstat(), lseek(), lstat(), mkdir(), open(),
opendir(), printf(), read(), readdir(), rename(), rewinddir(),
seekdir(), stat(), telldir(), truncate(), umask(), unlink(),
utime(), write()
```

18.1 INTRODUCTION

The kernel does the steady-state maintenance of a UNIX operating system, and the kernel provides services to user programs through the *system call interface* (SCI). This maintenance is achieved by using three techniques: *virtualization*, *concurrency*, and *persistence*. Virtualization allows devices, such as a single or multicore CPU, to act as if it were many CPUs acting simultaneously in a time-sharing manner. Concurrency allows multithreaded or multiprocess programs to access the virtualized resources of the hardware. Persistence allows data to be retained over time via mechanisms of I/O onto devices such as a solid state drive (SSD).

In this and the subsequent three chapters, we describe how various components of the SCI perform their roles in the realization of virtualization, concurrency, and persistence. This chapter deals with persistence. Chapter 19 deals with virtualization and concurrency. Chapter 20 deals with the communication aspect of virtualization and concurrency. Chapter 21 deals with the various practical issues related to virtualization, concurrency, and persistence.

In this chapter, we discuss the use of the API provided by the SCI for file handling. We assume that the reader is familiar with file handling through the use of the standard I/O library that is built around the ISO C standard specification and is available on multiple operating system platforms, including UNIX and Microsoft Windows. Our coverage of file handling is focused on performing data I/O with regular files. We also investigate how you can access and display a file's attributes by reading certain kernel data structures. We do not describe in detail the system calls for changing file attributes, but explain them briefly so you can explore them on your own. Also, we do not cover the details of directory handling. However, we provide a summary of the relevant system calls and library functions for performing such chores. Our coverage of the various topics is closely linked with the underlying kernel data structures and operating system concepts where appropriate. For the compilation of our sample C programs we use `gcc48`, the standard compiler command on PC-BSD. If you are using Solaris, you can use the `gcc` command.

18.2 WHAT IS SYSTEM PROGRAMMING?

Application programming is the skill of writing programs to provide services to users, including word processing, text and graphics editing, video processing, voice streaming, and Internet services such Web browsing. *System programming* is the ability of writing the kernel code to manage system hardware, including main memory and disk space management, disk formatting and defragmentation, CPU scheduling, and management of I/O devices through device drivers. Writing compilers or any part of the operating system code also falls in the realm of system programming. Whereas application programming requires users to be savvy in using language libraries, system programming requires a high degree of understanding of the computer hardware, assembly language, and a language like C.

In UNIX jargon, system programming uses the system call interface in order to access hardware resources such as the CPU, disk and files, main memory, and the status of processes and files. Such programs include, for instance, a shell, a language assembler or compiler, tools for providing information about a computer's hardware resources including the usage of CPU, disk, and main memory, and programs that provide the status of software resources such as processes and files.

18.3 ENTRY POINTS INTO THE OS KERNEL

There are several reasons for control to transfer from a user process to the OS kernel. These reasons collectively fall into four categories, known as four *entry points* into the kernel, as shown in Figure 18.1. Because the purpose of these entry points is to invoke the relevant pieces of codes in the kernel to provide different services, they are also known as *service points* into the kernel.

Two of these entry points, *trap* and *interrupt*, are caused by computer hardware. An *interrupt* is a "signal" that a peripheral hardware device sends to the CPU in order to get its attention. For example, after a disk controller has finished reading a disk block (or cluster), it needs to inform the CPU about it. For this purpose, it generates an interrupt and the kernel takes over control in order to execute the code to service this interrupt, called the *interrupt service routine* (ISR) for this interrupt.

Whereas a peripheral device generates an interrupt, the CPU itself generates a *trap* in order to handle an exception in the code being executed. There are several reasons for the CPU to do so, including execution of an illegal instruction (instruction not in the instruction set of the CPU), a potential run-time error like a divide-by-zero situation, and an instruction trying to access a main memory area outside the process address space.

A *signal* in UNIX jargon is a mechanism that allows interruption of a process. In the computer science literature, it is also known as *software interrupt*. In Section 13.4, we discussed this topic in a fair amount of detail. However, the discussion was focused on shell-level handling of signals. Table 13.1 shows some of the commonly used signals and their

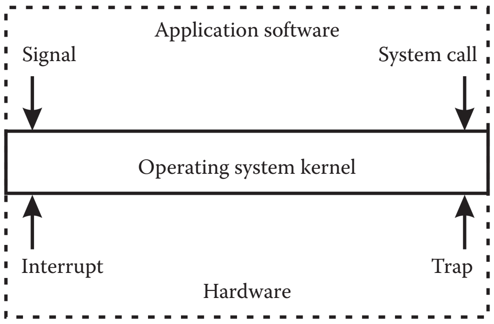


FIGURE 18.1 Entry points into the operating system kernel.

purpose. Note that some of the signals are generated through keystrokes but most are software generated. We discuss signals and signal handling from a system programmer's point of in detail in Chapter 19.

The fourth entry point, *system call*, is the topic of discussion in the remainder of this and next three chapters.

18.4 FUNDAMENTALS OF SYSTEM CALLS

The SCI is a mechanism that allows a user process to execute a piece of code in the operating system kernel. Processes use system calls to invoke services that allow access to hardware devices and kernel resources (code and data) that processes are not allowed to access otherwise, particularly in a multiuser, time-sharing system. This limitation is necessary in order to protect the resources (data and code) belonging to the kernel and other users from accidental or judicious access by a user. The system call interface ensures such protection and security.

18.4.1 What Is a System Call?

A user process is not allowed to have direct access to computer hardware such as a disk drive and kernel data structures in order to ensure that resources belonging to other users and the kernel remain protected. However, a user does need to access resources that he/she owns on the system such as files, as well as information about various system resources (hardware and software) such as CPU utilization or any number of processes running on the system. A system call is a mechanism that allows a process to perform privileged tasks that it is not allowed to perform by directly accessing (reading or writing) an I/O device or executing a piece of kernel code.

Thus, a system call is an entry point into the UNIX kernel code. In other words, it is a way for a process to execute a piece of code in the kernel, ensuring protection of resources that do not belong to the process. UNIX offers this mechanism to provide several types of services, including the following:

- Opening a file
- Reading and writing file data and attributes
- Accessing file attributes
- Setting file attributes
- Obtaining information about system hardware and operating system
- Creating a process
- Creating a channel for communication between processes
- Getting attention of a process and having it perform a particular task
- Obtaining information about the processes currently running on the system
- Creating and terminating processes

- Stopping processes
- Making processes wait for different events
- Creating communication channels for processes to communicate with each other
- Accessing process attributes
- Accessing utilizations of hardware resources such as memory and CPU utilization

These service points are provided through *wrapper* (library) functions, one for each system call. System programmers use these library functions to invoke relevant services.

18.4.2 Types of System Calls

There are several types of system calls dealing with the various aspects of the UNIX services. We categorize them as follows:

- Process control
- File management
- Device management
- Information maintenance
- Communications

In this and subsequent chapters on system programming in this book, we will primarily cover some commonly used system calls related to process control, file management, and interprocess communication. We discuss system calls related to file management in this chapter. In Chapters 19 and 20, we describe system calls for process management and interprocess communication.

EXERCISE 18.1

Browse the Web for manual pages on UNIX system calls and list two system calls each for the five types listed in Section 18.4.2 along with their purpose.

18.4.3 Execution of a System Call

Although not necessary to the learning of system programming and becoming good at it, knowing the low-level details of system call execution enhances your understanding of the process an operating system goes through to execute a system call. These steps are hardware dependent and vary from system to system. Here is an example sequence of steps.

1. The user program makes a call to a library function that acts as a wrapper for the system call.

2. The library function:
    a. Puts the parameters of the system call on the process stack.
    b. Passes a *call number* (N) that uniquely identifies the system call via a register or stack, or as a parameter to the `trap` function (see next bullet).
    c. Executes a CPU instruction, called the `trap` instruction on some CPUs (such as the Motorola microprocessors), to switch the CPU mode from *user* to *kernel/system* and transfer control to the `syscall()` function in the UNIX kernel.
3. The `syscall()` function is the system call handler and performs the following tasks:
    a. Identifies the system call invoked based on the call number.
    b. Copies the appropriate number of parameters from the user stack to the kernel stack.
    c. Uses the call number to index the *dispatch table* (which contains pointers to service routines for system calls) to execute the code for the requested service.
4. The code for the respective service executes and:
    a. The return value is placed in one or two registers for transferring it to the caller process (the return value is –1 in case of failure).
    b. In case of failure, the appropriate error code is placed in a register.
5. Control transfers back to the library code, which performs the following tasks:
    a. In case of error, the error code, placed in a register in 4(ii), is saved at a known location (for `errno`).
    b. The instruction following the `trap` instruction executes.
6. Execution of the user program continues.

Figure 18.2 gives a pictorial view of these steps.

18.5 FILES: THE BIG PICTURE

Recall that nearly everything in UNIX is a sequence of bytes: files, directories, I/O devices, network cards, and so on. Further, UNIX treats all files as streams of bytes. Thus, it treats all files consistently. We discussed the structure of a directory entry in UNIX, comprising of filename and inode number, and the concept of file descriptors in Chapter 4. We can access a file or its attributes without opening it through the file's inode. After opening a file using the corresponding system call, its contents are accessed through its file descriptor.

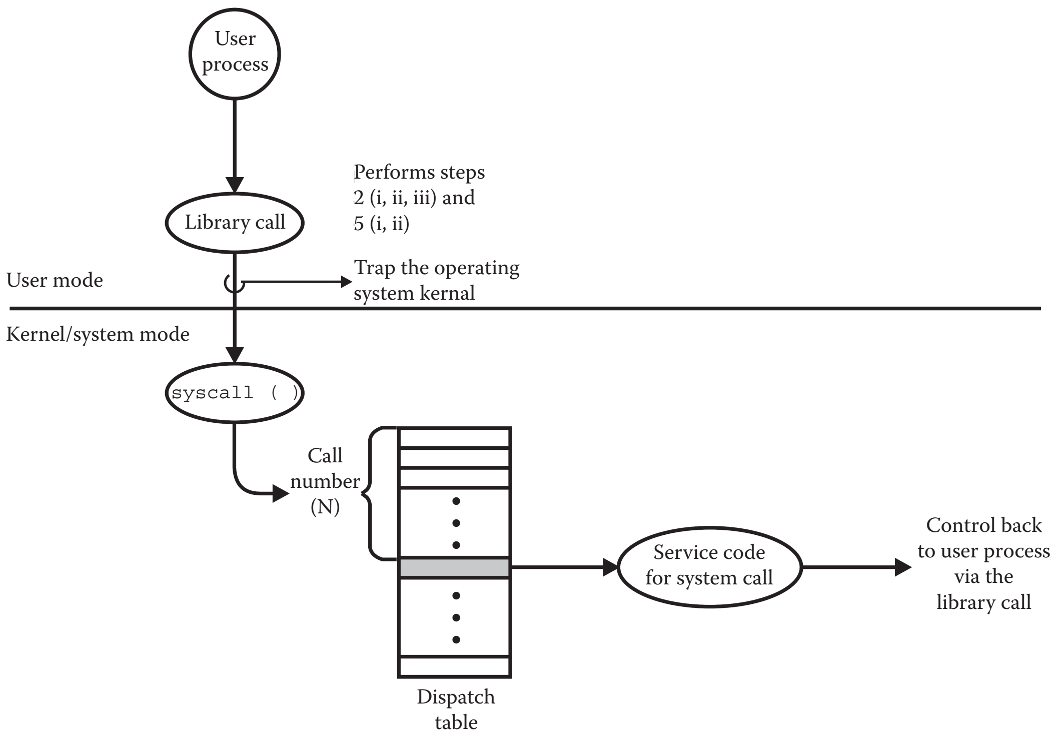

User process
Library call
Performs steps 2 (i, ii, iii) and 5 (i, ii)
User mode
Trap the operating system kernal
Kernel/system mode
syscall ( )
Call number (N)
Service code for system call
Control back to user process via the library call
Dispatch table

FIGURE 18.2 The sequence of steps required for the execution of a system call.

18.5.1 File Descriptors, File Descriptor Tables, File Tables, and Inode Tables

We discussed the concept of file descriptors in detail in Chapters 4 and 9. However, that discussion was focused on the use of standard file descriptors for I/O and error redirection for shell commands. In this chapter, we focus on the concept and use of file descriptors, including the standard file descriptors, from a system programmer's point of view.

File descriptors are non-negative integers starting with 0 and are used to index the *per-process file descriptor tables* (PPFDTs). We discussed this concept in Chapter 4 (Sections 4.7–4.9) and throughout most of Chapter 9, starting with Section 9.6. Figure 4.6 shows the relationship between PPFDTs, system-wide file tables (SFTs), inode tables, and a file's contents on a secondary storage.

Several UNIX system calls, including `open()`, `creat()`, `pipe()`, and `socket()`, return file descriptors for the file, pipe, and socket that they open or create. Communication with these objects using the `read()` and `write()` system calls is carried out through these descriptors. We discuss the `open()` and `creat()` system calls later in this chapter, and describe the `pipe()` and `socket()` calls in Chapters 19 and 20, respectively.

Processes perform data I/O and error output also through file descriptors for the three files that the UNIX kernel opens automatically for every process. As discussed in Chapter 4 and Chapter 9, these files are known as standard files: *standard input*, *standard output*, and *standard error*. Standard input is the default location from where a process takes its input. Standard output is the default location where a process's output goes to and standard error is the file where errors generated by a process are sent. The integer values for standard input, standard output, and standard error are 0, 1, and 2, respectively. Thus, the first descriptor returned by a system call is 3, by default. However, if you close one of the standard files, execution of the `open()`, `creat()`, `pipe()`, or `socket()` system call will return the first available descriptor, starting with 0.

18.5.2 Why Two Tables?

The UNIX PPFDT keeps track of all the files that a process has opened. The SFT keeps track of all the files open in a UNIX system at any given time and links the PPFDTs and the inode table, as shown in Figure 18.3.

Why did the designers of the UNIX kernel need to have the SFT? Why could they not have an entry in the PPFDT point directly to an entry in the inode table? The answer is that since UNIX allows a file to be opened multiple times simultaneously, it needs to keep track of multiple read/write file pointers (See Section 18.8.5). Thus an inode cannot be used to store all of the attributes of a file, and a separate data structure is needed where file pointers for files that have been opened multiple times simultaneously may be maintained. The SFT is an array (i.e., table) of such a data structure. Note that in Figure 18.3, a file has been opened in two different processes, P1 and P2. In P1, the file descriptor for the file is 3 and in P2 the descriptor is 5. Because the file has been opened twice, it has two entries in the SFT, both of which point to the inode of the file. Each entry in the SFT contains the current position of the read/write file pointer.

EXERCISE 18.2

Browse through the <**limits.h**> file on Solaris to determine the size of the PPFDT on your system. What is it? Clearly write down the exact definition as given in the header file.

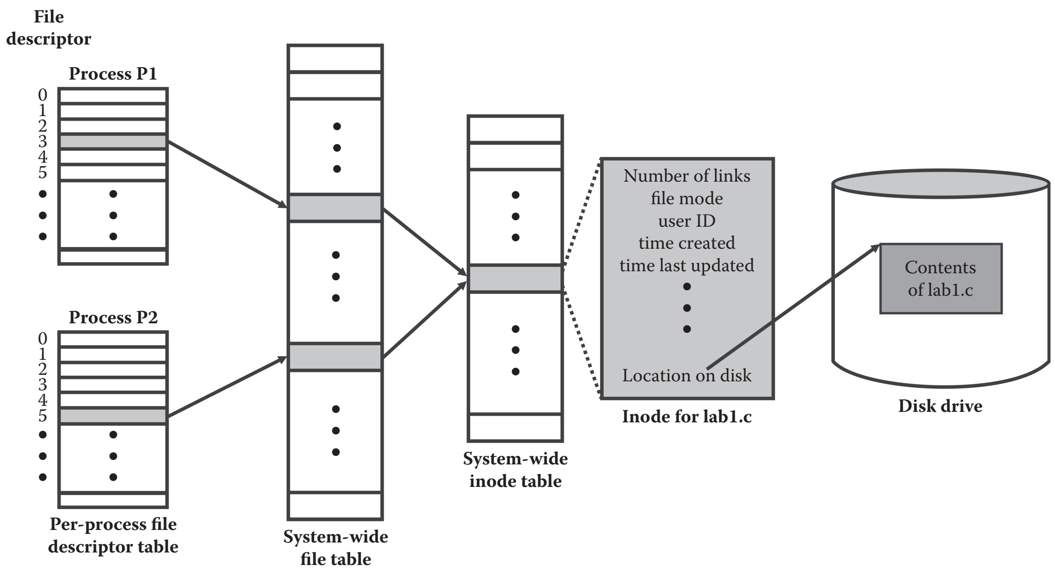


FIGURE 18.3 The relationship between PPFDT, SFT, inode table, and secondary storage.

EXERCISE 18.3

What is the size of the SFT on your system? How did you obtain your answer? Write down if your system is Solaris or FreeBSD.

18.6 FUNDAMENTAL FILE I/O PARADIGM

The basic file I/O paradigm in UNIX is *open*, *read*, *write*, and *close*. It means that a process must open a file before performing a data I/O (i.e., read/write) operation on it. If a file does not exist, it should be created and opened before data I/O can be performed on it. If a file is created using the `open()` system call (see Section 18.8.1 for details), it can be created and opened according to the access permission specified as a parameter. If, however, the file is created using the `creat()` system call (see Section 18.8.2), it must be explicitly opened with the `open()` system call for I/O. As I/O is being performed on a file, the position of the read/write file pointer may be changed in order to carry out the read/write operations at the desired byte position in the file. We discuss this issue in Section 18.8.5. A file does not have to be opened for reading or writing its attributes. We discuss this topic in Section 18.8.

18.7 STANDARD I/O VERSUS LOW-LEVEL I/O

The UNIX API provides two interfaces for file I/O, one via the Standard I/O library and the second through the SCI. Since the SCI resides immediately above the kernel and the language libraries are built on top of the SCI, I/O via the SCI is called *low-level I/O*. Since the C Standard Library (CSL) interface is built on top of the SCI, it is easy to use but slower than the SCI.

The low-level I/O interface via the SCI is more time efficient because of its direct interface with the UNIX kernel. However, it is cumbersome because the programmer needs to handle buffer issues such as buffer allocation/deallocation and choosing the right size block for I/O.

18.7.1 The C Standard Library

The CSL implementation is based on the ISO C standard specification that has been implemented on several operating systems, including UNIX, Linux, and Microsoft Windows. Thus, code written using this library is portable across operating systems. The library consists of a set of predefined and well-tested functions, constants, and header files. It provides for you the mechanisms to handle such tasks as character I/O to file I/O, from time-related functions to complex math functions, and from string operations to signal handling. The header files in this library contain hundreds of variable types, functions, and macros. Some of the definitions are duplicated in multiple files for the sake of completion. For example, NULL is defined in multiple header files. Table 18.1 lists the header files that comprise the CSL.

18.7.2 File Data I/O Using the C Standard Library

The Standard I/O library API is simple to use and handles the issues of buffer allocation and using appropriate size blocks for optimal I/O in terms of time (both CPU and clock

TABLE 18.1 Header Files for C Standard Library and Their Purpose

Header File	Purpose
`<assert.h>`	It contains a macro, called assert, that can be used to diagnose whether the assumptions made in your program are correct or not.
`<ctype.h>`	It contains a set of functions to test the attributes of characters (i.e., whether a character is a decimal digit, a control character, etc.). It also contains two functions to convert a lowercase character to uppercase, and vice versa.
`<errno.h>`	It contains the definition of the `errno` variable that is set by system calls and some library functions. It also contains a few macros that indicate different error codes.
`<float.h>`	It contains a set of constants and macros that deal with floating point values in a machine-dependent way.
`<limits.h>`	It contains the minimum and maximum values of different variable types, including `signed char`, `unsigned char`, `int`, `unsigned int`, `short int`, `unsigned short int`, `long`, and `unsigned long`.
`<locale.h>`	It contains currency symbols and date formats.
`<math.h>`	It contains math functions, including sin, cos, tan, sqrt, log, ceil, and floor.
`<setjmp.h>`	It contains a macro, a function, and a variable used to bypass the normal function call/return mechanism.
`<signal.h>`	It contains functions and macros for signal handling (discussed in Chapters 13 and 15).
`<stdarg.h>`	It contains a functions and macro to handle variable number of arguments in a function.
`<stddef.h>`	It contains variable types and macros, some of which appear in other header files too.
`<stdio.h>`	It contains variable types, functions, and macros for I/O.
`<stdlib.h>`	It contains variable types, functions, and macros for general-purpose tasks, including conversion from ASCII to float, random number generation, and dynamic memory allocation/deallocation.
`<string.h>`	It contains variable types, functions, and macros for handling character arrays (i.e., character strings).
`<time.h>`	It contains variable types, functions, and macros for handling date and time, including conversion from one format to another.

times). The functions in this library use the file I/O calls available in the SCI. The purpose of using the right size buffers and block sizes is to minimize the number of disk I/O operations by minimizing the number of `read()` and `write()` system calls (see Sections 18.8.3 and 18.8.4).

The Standard I/O library supports three types of *buffered I/O*: *fully buffered*, *line buffered*, and *unbuffered*. The ISO C standard requires that standard input and standard output be fully buffered for noninteractive devices. On the other hand, standard error is never fully buffered; it is normally unbuffered so that error messages are displayed at the earliest possible time.

When we open a file using the Standard I/O interface, we get back a pointer to an object `FILE`, called the *file pointer*. We perform all subsequent file operations using this pointer and the standard I/O library functions. When we open a file using the SCI, we get back an object of type `int`, called the *file descriptor*. We perform all subsequent I/O operations on the file using this descriptor and the relevant system calls. Another term used for both of these objects is *file handle*. In this book, we primarily deal with the low-level I/O interface.

When we open a file using the Standard I/O library interface, a *stream* is associated with the file. Depending on the character set used, a character can be represented using one or multiple bytes. Thus, stream I/O depends on the "width" of a character in terms of bytes and determines the "orientation" of the stream. When a stream is created it has no orientation and its orientation is set by the type of functions used for performing I/O with the stream. When we open a file with the `open()` system call, a *sequence of bytes* is associated with the file and all subsequent I/O takes place in terms of byte sequence.

We assume that the reader is familiar with file I/O using the Standard I/O library and do not discuss it any further.

18.7.3 Low-Level I/O in UNIX via System Calls

Table 18.2 shows UNIX system calls for file I/O along with their purpose. We will discuss most of these system calls in this chapter and the remaining in Chapter 19. As discussed in Section 18.4.3, these are in fact wrapper functions that eventually transfer control to the system call, `syscall()`, along with the respective call number and call parameters so that appropriate kernel function may be executed to serve the relevant system call. You need to include the **`<fcntl.h>`** file in order to be able to use the `open()` and `creat()` system calls. For the `close()` and `lseek()` system calls, you need to use the **`<unistd.h>`** file. Finally, for using `read()` and `write()` you need to include the **`<sys/types.h>`** and **`<unistd.h>`** files.

In this chapter, we discuss the system calls for performing the following operations on files: creating, opening, closing, reading, writing, preparing for random access, deleting a hard link to a file, and getting file attributes. You will get to practice some of the remaining calls in some of the programming exercises given at the end of the chapter.

TABLE 18.2 The UNIX System Calls for File I/O

Creating, Opening for I/O, Closing	
`creat()`	Create a file
`open()`	Open or create and open a file
`close()`	Close a file
Data I/O	
`read()`	Read from an open file
`write()`	Write to an open file
Creating and Removing Files (Hard Links)	
`link()`	Create a hard link to a file
`unlink()`	Remove a hard link to a file and if the resultant link count becomes zero, delete the file
`remove()`	Identical to `unlink()`
Setting Up for Random Access	
`lseek()`	Set file for random access

18.7.4 System Call Failure and Error Handling

A UNIX system call returns –1 on failure. Thus, while using a system call in your code, you must first check whether the call has failed and use appropriate code to handle it. In most cases, you would like to display an adequate error message to the user by using the C Standard Library function `perror()` and terminate the program execution by using the `exit()` system call.

EXERCISE 18.4

Browse through **<stdio.h>** and identify the number of functions and macros defined in it.

EXERCISE 18.5

List the Standard I/O library calls for character and file processing. Give a one-sentence description of each call.

18.8 FILE MANIPULATION

In this section, we discuss the UNIX system calls to access and manipulate file data and attributes. Our focus will be on the following operations:

- Opening a file
- Creating a file
- Reading file data
- Writing data to a file
- Positioning the read/write file pointer for random access
- Truncating a file
- Closing a file
- Checking the existence of a file
- Removing a file
- Obtaining and displaying file attributes

We will primarily focus on manipulating data in regular files. However, we discuss the access, display, and modification of file attributes for files of all types. We do not discuss the manipulation of directories, but briefly describe the system call interface for doing so. The system calls for handling sockets and FIFOs are described in detail in Chapters 20 and 21.

18.8.1 Opening and Creating a File

You can use the `open()` system call to open an existing file for reading, writing, or performing both operations. If a file does not exist, you can create it and open it for writing by using the `creat()` system call. Obviously, reading from a newly created (empty) file does not make sense. You can also create a file by specifying appropriate flags in the `open()` call to request the kernel to create the file if it does not exist, and set it for reading, writing, or both. The following are the brief descriptions of the two system calls.

```
#include <fcntl.h>
int open(const char path, int flags, ... /* mode_t mode */ );
```
Success: Non-negative integer, called a file descriptor
Failure: –1 and kernel variable `errno` set to indicate the type of error

```
#include <fcntl.h>
int creat(const char path, mode_t mode);
```
Success: Non-negative integer, called a file descriptor
Failure: –1 and kernel variable `errno` set to indicate the type of error

The third argument (`...`) in the prototype of the `open()` call means that the number and type of remaining arguments may vary. The `mode` argument in the `creat()` call is used to specify the access permissions (see Chapter 5) for the newly created file. Both calls return a non-negative integer called the file descriptor of the opened file, which can be used to perform the desired I/O operations by using the `read()` and `write()` system calls (discussed later), change the position of the read/write file pointer for random access, and close the file using the `close()` system call. The returned descriptor is the first unused descriptor in the PPFDT, starting with descriptor 0. If the file opens successfully, the read/write file pointer is set to the beginning of the file. It means that the first read or write operation starts at the first byte of the file. Both calls return –1 on failure.

Bitwise `OR`ing two or more constants listed in the **<fcntl.h>** file forms the flags argument. Table 18.3 shows some of the commonly used flags used for file I/O and their meanings.

The `open()` call may fail for several reasons. Some of the commonly occurring reasons for the call to fail for regular files are given in Table 18.4, along with the corresponding symbolic values saved in the `errno` variable.

Since the `creat()` system call creates a file and, by default, opens it for writing only, the following calls are equivalent to each other:

```
creat(pathname, mode);
open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

TABLE 18.3 Commonly Used Flags for File I/O

Flag	Meaning
`O_RDONLY`	Open the file for reading only.
`O_WRONLY`	Open the file for writing only.
`O_RDWR`	Open the file for reading and writing.
`O_CREAT`	Create the file if it does not exist. If you use this option, you must specify the access permissions for the newly created file as the third argument, `mode`.
`O_APPEND`	Each write operation appends at the end of file.
`O_TRUNC`	If the file exists, its length is set to 0. Previous file contents are not accessible anymore.

TABLE 18.4 A few Common Reasons for the Failure of the `open()` System Call

Reason for Failure	Value of `errno`
A component in the **path** does not exist.	`ENOENT`
A component in the **path** is not a directory.	`ENOTDIR`
The named file is a directory.	`EISDIR`
Search permission is not set on a component in the **path**.	`EACCESS`
The process has already opened the maximum number of files allowed by the system, i.e., the PPFDT is already full.	`EMFILE`
The system has already reached the limit of the maximum number of files that may be opened on it simultaneously (i.e., the system-wide file table is full).	`EMFILE`
The file opening operation was interrupted by a signal.	`EINTR`
A component in **path** exceeds the file name size limit (255 characters) or the entire path exceeds the 1023 characters.	`ENAMETOOLONG`
The named file is a special file, but the device associated with the file does not exist.	`ENXIO`
`O_CREAT` flag is specified to create a file but the file system is read only.	`EROFS`
`O_CREAT` flag is specified and error occurred while creating an inode for the new file or making the directory entry for the file to be created.	`EIO`
`O_CREAT` and `O_EXCL` flags are specified and the file specified in `path` exists.	`EEXIST`

The `open()` system call follows a symbolic link and accesses the data or attributes of the file that the link points to. If you want to read the data or attributes of a symbolic link, you must use the `readlink()` system call. Similarly, the `creat()` system call can create only an ordinary file. If you want to create a symbolic link, you should use the `symlink()` system call. You should browse through the man pages for these calls to learn more about them.

EXERCISE 18.6

Give the prototypes of the `symlink()` and `readlink()` system call. Briefly describe each parameter for these calls.

18.8.2 Closing a File

You can close a file by using the `close()` system call, which takes a file's descriptor as an argument and deallocates the PPFDT slot corresponding to the given descriptor. The following is a brief description of the call.

```
#include <unistd.h>
int close(int fd);
```

Success: `0`
Failure: –1 and kernel variable `errno` set to indicate the type of error

The `close()` system call may fail for several reasons. Two commonly occurring reasons are given in Table 18.5, along with the symbolic values of the `errno` variable corresponding to these errors.

TABLE 18.5 Commonly Occurring Reasons for the Failure of the `close()` System Call

Reason for Failure	**Value of** `errno`
`fd` is not an active descriptor, i.e., does not correspond to an open file.	`EBADF`
The file-closing operation was interrupted by a signal.	`EINTR`

If an open file was being referenced by more than one process—that is, if entries in multiple PPFDTs are pointing to an entry in the SFT—then closing the file would only deallocate the entry in the relevant PPFDT and decrement the reference count by one. A file is closed only when this reference count becomes zero. When a process terminates, the kernel automatically closes all of its open files.

In the following example, we open the file passed as the only command line argument, display the file descriptor of the file, and close the file. The compilation and execution of the example program shows the program output. Note that the default compiler commands are `gcc` and `gcc48` on Solaris and PC-BSD. We show compilation of programs on the BSD system. We use the compiler command with the –w option in order to suppress the warning messages.

```
% cat open_close.c
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
       int fd;

       /* Open file */
       if (argc == 1) {
              printf("No file specified as command line
                     argument.\n");
              exit(1);
       }
       if ((fd = open(argv[1], O_RDWR)) == -1) {
              perror("File opening");
              exit(1);
       } else
              printf("The file descriptor is %d.n",fd);
       /* Close file */
       if (close(fd) == -1) {
              perror("File closing");
              exit(1);
       }
       else
              printf("File closed successfully.\n");
       exit(0);
}
```

```
% gcc48 -w open_close.c
% ./a.out foobar
The file descriptor is 3.
File closed successfully.
%
```

Note that this is the first (and the only) file that the process has opened. Since the kernel has already opened the standard files for this process, the kernel allocated file descriptor 3 for when the program opened **foobar**.

We now modify the program slightly so that it first closes standard input and then opens the file passed to it as a command line argument. In order to do so, we only need to insert the following line before the open() system call:

```
close(0);
```

Here is the updated program:

```
% cat open_close.c
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
       int fd;

       /* Open file */
       if (argc == 1) {
              printf("No file specified as command line
                     argument.\n");
              exit(1);
       }
       close(0); /* Close standard input */
       if ((fd = open(argv[1], O_RDWR)) == -1) {
              perror("File opening");
              exit(1);
       } else
              printf("The file descriptor is %d.\n",fd);
       /* Close file */
       if (close(fd) == -1) {
              perror("File closing");
              exit(1);
       }
       else
              printf("File closed successfully.\n");
       exit(0);
}
% gcc48 -w open_close.c
% ./a.out foobar
```

```
The file descriptor is 0.
File closed successfully.
%
```

As you can see, as expected, the kernel allocated file descriptor 0 to the newly opened file **foobar**.

EXERCISE 18.7

Compile and run the programs given in Sections 18.8.1 and 18.8.2. Do they work as intended?

18.8.3 Reading from a File

Once a file has been opened using the `open()` system call, you can read its contents by using the `read()` system call. Here is a brief description of the `read()` call.

```
#include <sys/types.h>
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t nbytes);
```

Success: 0
Failure: –1 and kernel variable `errno` set to indicate the type of error

This call tries to read `nbytes` bytes from the file with the file descriptor `fd` into the main memory area pointed to by `buf`. The data is read starting with the current location of the read/write file pointer. Upon completion, the `read()` call returns the number of bytes actually read, which may be less than `nbytes`. The file pointer advances by the number of bytes actually read. For a regular file, the call guarantees reading `nbytes` if the file has these many bytes left before the end-of-file (EOF). However, with other files, such as pipes or sockets (see Chapters 19 and 20), this is not guaranteed. The `read()` system call returns 0 when it encounters EOF.

On failure, `read()` returns –1 and `errno` is set to indicate the reason for failure. Table 18.6 shows some common reasons for the `read()` call to fail for regular files.

TABLE 18.6 Some Common Reasons for the `read()` System Call to Fail

Reason for failure	**Value of `errno`**
The `fd` argument is not a valid descriptor for reading.	`EBADF`
The `buf` argument points to a memory location outside the process address space.	`EFAULT`
An I/O error occurred while reading from the file system.	`EIO`
The file reading operation was interrupted.	`EINTR`
The pointer associated with the `fd` argument is negative.	`EINVAL`
The `nbytes` value is greater than `INT_MAX`, i.e., the maximum value of an integer.	`EINVAL`

18.8.4 Writing to a File

Once a file has been opened using the open() or creat() system call, you can write to it using the write() system call. Here is a brief description of the write() call:

```
#include <sys/types.h>
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t nbytes);
```

Success: 0
Failure: –1 and kernel variable errno set to indicate the type of error

This call tries to write nbytes bytes from the main memory area pointed to by buf to the file with the file descriptor fd. The data is written starting with the current location of the read/write file pointer. Upon completion, the write() call returns the number of bytes actually written, which may be less than nbytes. The file pointer advances by the number of bytes actually written. For a regular file, the call guarantees writing nbytes if the disk is not full or the file size has not exceeded the maximum file size supported by the UNIX system. However, with other files, such as sockets (see Chapter 20), this is not guaranteed.

On failure, write() returns –1 and errno is set to indicate the reason for failure. Table 18.7 shows some common reasons for the read() call to fail for regular files.

We now enhance the open _ close.c program and convert it into a file copy program, cpy, with the following syntax:

```
cpy source target
```

The program copies the **source** file, an existing file, to the **target** file, a nonexistent file. The compilation and execution of the example program shows the program output.

```
% cat cpy.c
#include <fcntl.h>
#include <unistd.h>

#define SIZE 512

void closefd(int);
```

TABLE 18.7 Some Common Reasons for the write() System Call to Fail

Reason for failure	**Value of errno**
The fd argument is not a valid descriptor for writing	EBADF
The fd argument is associated with a negative pointer.	EINVAL
File size limit for the process or the maximum file size limit has reached.	EFBIG
The file system containing the file is full.	ENOSPC
The user's quota of disk blocks on the file system containing the file has been exhausted.	EDQUOT
An I/O error occurred while reading from the file system.	EIO
The file reading operation was interrupted.	EINTR
The nbytes value is greater than INT_MAX, i.e., the maximum value of an integer.	EINVAL

```
int main(int argc, char *argv[])
{
       int rfd, wfd, nr, nw;
       char buf[SIZE];

       if (argc != 3) {
              printf("Inappropriate number of command line
                     arguments.\n");
              exit(0);
       }
       /* Open the source file for reading */
       if ((rfd = open(argv[1], O_RDONLY)) == -1) {
              perror("Source file opening");
              exit(1);
       }
       /* Open the destination file for writing */
       if ((wfd = open(argv[2], O_CREAT | O_WRONLY | O_TRUNC,
            0666)) == -1) {
              perror("Creation and opening of destination file");
              exit(1);
       }
       while ((nr = read(rfd, buf, sizeof(buf))) != 0) {
              if (nr == -1) {
                     perror("File reading");
                     exit(1);
              }
              nw = write(wfd, buf, nr);
       }
       closefd(rfd);
       closefd(wfd);
       return(0);
}
void closefd(int fd)
{
       if (close(fd) == -1) {
              perror("File closing");
              exit(1);
       }
}
% gcc48 -w cpy.c
% ./a.out cpy.c cpy_bak.c
% ls -l
total 18
-rwxr-xr-x  1 sarwar  faculty  7960 Jan 29 23:27 a.out
-rw-r--r--  1 sarwar  faculty   788 Jan 29 23:27 cpy.c
-rw-r--r--  1 sarwar  faculty   788 Jan 29 23:29 cpy_bak.c
%
```

EXERCISE 18.8

Compile and run the programs given in Section 18.8.4. Do they work as intended?

EXERCISE 18.9

Remove the `O _ TRUNC` flag in the second `open()` system call. What is its effect? Explain your answer.

18.8.5 Positioning the File Pointer: Random Access

You can use `lseek()` to change the position of the read/write file pointer. This service allows you to access file data randomly. Here is a brief description for the call.

```
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```
Success: `0`
Failure: –1 and kernel variable `errno` set to indicate the type of error

The call sets the position of the read/write file pointer for the file with descriptor `fd` to `offset` according to the value of the `whence` argument. The repositioning of the descriptor is done according to Table 18.8.

On successful completion, `lseek()` returns the resulting offset location in bytes from the beginning of the file. It returns -1 on failure and `errno` is set to indicate the reason for error. Thus, if you want to append data to a file, you should set the file pointer to the end of the file by using the following statement:

```
n = lseek(fd, 0L, SEEK_END);
```

Note that this call sets the file pointer to 0 bytes away from the current EOF and returns the new position (in bytes) of the file pointer. The new position also indicates the size of the file in bytes. Thus, when `lseek()` returns, `n` contains the size of the file (in bytes).

TABLE 18.8 Repositioning of the Read/Write File Pointer for Random Access of File Data

The Value of `whence`	Position of the Read/Write Pointer
`SEEK_SET`	The position/offset is set to `offset` bytes from the beginning of the file.
`SEEK_CUR`	The position/offset is set to `offset` bytes from the current location of the pointer.
`SEEK_END`	The position/offset is set to `offset` bytes from the end of the file (i.e., to the size of the file plus `offset`).
`SEEK_HOLE`	The position/offset is set to the start of the next hole greater than or equal to `offset`.
`SEEK_DATA`	The position/offset is set to the start of the next non-hole (i.e., data region) greater than or equal to `offset`.

You can determine the current position of the file pointer by using the following statement. When the call returns, `n` contains the current position (in bytes) of the file pointer from the beginning of the file.

```
n = lseek(fd, 0L, SEEK_CUR);
```

This call only works for file objects that are capable of seeking. The use of `lseek()` for other types of files fails and `errno` is set to indicate this reason. Three such file types are pipe, FIFO, and socket, and device files not capable of seeking, such as keyboard. Table 18.9 shows some common reasons for the `lseek()` call to fail.

The `lseek()` call allows you to position the file pointer beyond the EOF. If data is written at this point later on, a *hole* is created between the previous EOF and the position of the file pointer after `lseek()`. The hole contains null bytes—that is, bytes containing zeros. Figure 18.4 shows the state of a file with holes.

The `create _ holes.c` program creates three holes of 512 bytes each in the file passed to it as a command line argument. After the sample run of the program, we use the `ls -l data` command to display the size of the data file with holes. We use the `tail data | od` commands to show the contents of the original file and after holes have been inserted in it as octal dumps (using the `od` command). Note that asterisks (*) are used in the octal dump to show a series of zeros. The count of zeros in one hole will come out to be, as expected, 512. Note that you cannot view the holes with the `tail data` command because it does not display null bytes.

TABLE 18.9 Some Common Reasons for the `lseek()` System Call to Fail

Reason for Failure	**Value of `errno`**
The `fd` argument is not an open file descriptor.	`EBADF`
The `fd` argument is associated with a file that it is not capable of seeking: pipe, FIFO, or socket.	`ESPIPE`
The `whence` argument or the resulting offset would be negative for a noncharacter special file.	`EINVAL`
The resulting offset cannot be represented in `off_t`.	`EOVERFLOW`
There is no data region (`SEEK_DATA`) or hole region (`SEEK_HOLE`) beyond `offset`	`ENXIO`

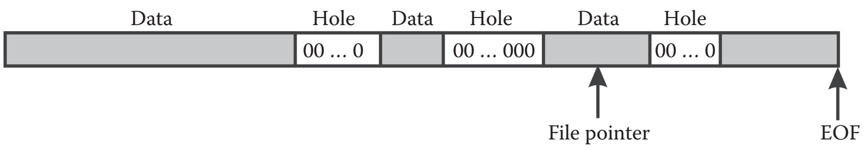


FIGURE 18.4 A UNIX file with three holes of different sizes.

```
% cat create_holes.c
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#define HSIZE     512L

int main(int argc, char *argv[])
{
       int fd, i, n, nholes, nw;
       char *buf="Random data for creating holes.";

       /* Open the given file */
       if (argc != 3) {
              printf("Inappropriate number of command line
                     arguments.\n");
              exit(0);
       }
       nholes = atoi(argv[2]);
       if ((fd = open(argv[1], O_WRONLY)) == -1) {
              perror("File opening");
              exit(1);
       }
       for (i=0; i<nholes; i++) {
              n = lseek(fd, HSIZE, SEEK_END);
              nw = write(fd, buf, strlen(buf));
       }
        if (close(fd) == -1) {
                perror("File closing");
                exit(1);
        }
       return(0);
}
% man bash > data
% ls -l data
-rw-r--r-- 1 sarwar faculty 367470 Jan 30 09:53 data
% tail data | od
0000000  020040 020040 020040 070040 071141 067145 064164 071545
0000020  071545 020040 067564 063011 071157 062543 020040 072151
0000040  020040 067151 067564 020011 020141 072563 071542 062550
0000060  066154 020054 064167 061551 020150 060555 020171 062542
0000100  071440 067564 070160 062145 060440 020163 005141 020040
0000120  020040 020040 072440 064556 027164 005012 020040 020040
0000140  020040 040440 071162 074541 073040 071141 060551 066142
0000160  071545 066440 074541 067040 072157 024040 062571 024564
0000200  061040 004545 074145 067560 072162 062145 005056 020012
0000220  020040 020040 020040 064124 071145 020145 060555 020171
0000240  062542 067440 066156 020171 067157 020145 061541 064564
```

```
0000260  062566 061440 070157 067562 062543 071563 060440 020164
0000300  020141 064564 062555 005056 005012 043412 052516 041040
0000320  071541 020150 027064 004463 004411 030062 032061 043040
0000340  061145 072562 071141 020171 004462 004411 020040 020040
0000360  020040 041040 051501 024110 024461 000012
0000373
% gcc48 -w create_holes.c
% ./a.out data 3
% ls -l data
-rw-r--r-- 1 sarwar  faculty  369099 Jan 30 09:54 data
% tail data | od
0000000  020040 020040 020040 072440 064556 027164 005012 020040
0000020  020040 020040 040440 071162 074541 073040 071141 060551
0000040  066142 071545 066440 074541 067040 072157 024040 062571
0000060  024564 061040 004545 074145 067560 072162 062145 005056
0000100  020012 020040 020040 020040 064124 071145 020145 060555
0000120  020171 062542 067440 066156 020171 067157 020145 061541
0000140  064564 062566 061440 070157 067562 062543 071563 060440
0000160  020164 020141 064564 062555 005056 005012 043412 052516
0000200  041040 071541 020150 027064 004463 004411 030062 032061
0000220  043040 061145 072562 071141 020171 004462 004411 020040
0000240  020040 020040 041040 051501 024110 024461 000012 000000
0000260  000000 000000 000000 000000 000000 000000 000000 000000
*
0001240  000000 000000 000000 000000 000000 000000 051000 067141
0001260  067544 020155 060544 060564 063040 071157 061440 062562
0001300  072141 067151 020147 067550 062554 027163 000000 000000
0001320  000000 000000 000000 000000 000000 000000 000000 000000
*
0002300  000000 000000 000000 000000 000000 000000 060522 062156
0002320  066557 062040 072141 020141 067546 020162 071143 060545
0002340  064564 063556 064040 066157 071545 000056 000000 000000
0002360  000000 000000 000000 000000 000000 000000 000000 000000
*
0003340  000000 000000 000000 000000 000000 051000 067141 067544
0003360  020155 060544 060564 063040 071157 061440 062562 072141
0003400  067151 020147 067550 062554 027163
0003412
% tail data
       unit.

       Array variables may not (yet) be   exported.

       There may be only one active coprocess at a time.

GNU Bash 4.3                  2014 February 2          BASH(1)
Random data for creating holes.Random data for creating holes.
Random data for creating holes.%
```

EXERCISE 18.10

Compile and run the program given in Section 18.8.5. Does it work as intended?

EXERCISE 18.11

Change the hole size to 2K. Compile and run the program. Verify that it works correctly.

18.8.6 Truncating a File

Truncating a file means chopping off contents from the tail of the file. The use of the `open()` system call with the `O _ TRUNC` flag, as shown in Section 18.8.4, is a special case of truncation where the file size is reduced to zero and the read/write file pointer and EOF are set to 0. You can use the `truncate()` or `ftruncate()` system calls to truncate a file. Here are the brief descriptions of these commands.

```
#include <unistd.h>
int truncate(const char *path, off_t length);
int ftruncate(int fd, off_t length);
```
Success: 0
Failure: –1 and kernel variable `errno` set to indicate the type of error

These functions truncate an existing file to `length` bytes. The difference between the functions is that `truncate()` works with a pathname and `ftruncate()` with a file descriptor. If `length` is smaller than the existing length of the file, the contents of the file beyond `length` bytes are not accessible anymore. If `length` is greater than the current file size, the file size is increased to `length` and the space between the previous EOF and new EOF is filled with zeros and becomes a hole (See Section 18.8.5).

The **truncate.c** program takes two command line arguments: a file and a number. It displays the size of the file by using the `lseek()` system call, truncates the file to the size specified as the second argument by using the `truncate()` system call, and displays the size of the truncated file. The sample run shows the sizes of the file, called data, before and after truncation. Note that the size of the input file is also confirmed with the `ls –l data` command. The `tail` commands are used to show the tails of the **data** file before and after truncation.

```
% cat truncate.c
#include <fcntl.h>
#include <unistd.h>

#define HSIZE 512L

int main(int argc, char *argv[])
{
        int fd, newfilesize, size_o, size_t;

        /* Open the given file */
        if (argc != 3) {
                printf("Inappropriate number of command line
                        arguments.\n");
                exit(0);
        }
        newfilesize = atoi(argv[2]);
        if ((fd = open(argv[1], O_WRONLY)) == -1) {
                perror("File opening");
                exit(1);
        }
        size_o = lseek(fd, 0, SEEK_END); /* Size of original file */
        if (truncate(argv[1], newfilesize) == -1) {
                perror("Truncating file");
                exit(1);
        }
        size_t = lseek(fd, 0, SEEK_END); /* Size of truncated file */
        printf("The size of the original file is %d bytes.\n",
                size_o);
        printf("The size of the truncated file is %d bytes.\n",
                size_t);
        if (close(fd) == -1) {
                perror("File closing");
                exit(1);
        }
        return(0);
}
```

```
% man cat > data
% ls -l data
-rw-r--r-- 1 sarwar faculty 4083 Jan 31 14:15 data
% tail data
BUGS
     Because of the shell language mechanism used to perform
     output redirection, the command "cat file1 file2 > file1"
     will cause the original data in file1 to be destroyed!

     The cat utility does not recognize multibyte characters when
     the -t or -v option is in effect.

FreeBSD 11.0       January 29, 2013         FreeBSD 11.0

% gcc48 -w truncate.c
% ./a.out data 1024
The size of the original file is 4083 bytes.
The size of the truncated file is 1024 bytes.
% tail data
```

```
    The options are as follows:

    -b     Number the non-blank output lines, starting at 1.

    -e     Display non-printing characters (see the -v option), and
           display a dollar sign ('$') at the end of each line.

    -l     Set an exclusive advisory lock on the standard output
           file descri%
```

EXERCISE 18.12

Compile and run **truncate.c**. Does it work as intended?

18.8.7 Removing a File

You can use the `unlink()` system call to remove a file from the file system structure. The following is a brief description of the call.

```
#include <unistd.h>
int unlink(const char *path);
```
Success: `0`
Failure: –1 and kernel variable `errno` set to indicate the type of error

This system call essentially decrements by one the hard link count for the file stored in its inode. If the resultant link count becomes zero and no process has the file open, the file is removed from the file system. If link count becomes zero but one or more processes have the file open, the removal of the file is delayed until the reference count for the file becomes zero. If the link count is not zero, the file remains in the file system structure and its directory entry is removed. Once a file has been removed from the file system, its directory entry, inode, and all other kernel resources associated with the file are returned to the system for recycling.

On successful completion, `unlink()` returns 0. It returns –1 on failure and `errno` is set to indicate the reason for failure. Table 18.10 shows some common reasons for the `unlink()` call to fail for regular files.

The **create_holes_delete.c** program shown in the following is an updated version of the **create_holes.c** program discussed in Section 18.8.5. The program saves the size of the original file, creates holes in it, displays the sizes of the original file and the file with holes, and then removes the file with holes. The program determines the sizes of the original and new file (with holes) by using the `lseek()` system call. The sample run shows the sizes of original file and the file with holes. The `ls -l  data` command shows that the program did remove the file with holes.

```
% cat create_holes_delete.c
#include <fcntl.h>
#include <unistd.h>
```

TABLE 18.10 Some Common Reasons for the `unlink()` System Call to Fail

Reason for Failure	**Value of** `errno`
The named file is a directory.	EISDIR
The file specified in **path** does not exist.	ENOENT
A component in **path** is not a directory.	ENOTDIR
Search permission is denied on a component of **path**.	EACCES
Write permission does not exist on the directory containing the file/link to be removed.	EACCES
A component in **path** exceeds the file name size limit (255 characters) and the entire path exceeds the 1023 characters.	ENAMETOOLONG
An I/O error occurred while deleting the directory entry for the file or deallocating file's inode.	EIO
The file to be removed resides on a read-only file system or device.	EROFS

```
#include <string.h>

#define HSIZE 512L

int main(int argc, char *argv[])
{
       int fd, i, n, nholes, nw, size_o, size_h;
       char *buf="Random data for creating holes.";

       /* Open the given file */
       if (argc != 3) {
              printf("Inappropriate number of command line
                     arguments.\n");
              exit(0);
       }
       nholes = atoi(argv[2]);
       if ((fd = open(argv[1], O_WRONLY)) == -1) {
              perror("File opening");
              exit(1);
       }
       size_o = lseek(fd, 0, SEEK_END); /* Size of original file */
       for (i=0; i<nholes; i++) {
              n = lseek(fd, HSIZE, SEEK_END);
              nw = write(fd, buf, strlen(buf));
       }
       size_h = lseek(fd,0,SEEK_END); /* Size of file with holes */
       if (close(fd) == -1) {
               perror("File closing");
               exit(1);
       }
       printf("The size of the original file is %d bytes.\n",
              size_o);
       printf("The size of the file with holes is %d bytes.\n",
              size_h);
       unlink(argv[1]);
       return(0);
}
% man bash > data
% ls -l data
-rw-r--r-- 1 sarwar faculty 367470 Jan 30 09:59 data
% gcc48 -w create_holes_delete.c
% ./a.out data 3
The size of the original file is 367470 bytes.
The size of the file with holes is 369099 bytes.
% ls -l data
ls: data: No such file or directory
%
```

The `remove()` system call is equivalent to the `unlink()` system call for files.

EXERCISE 18.13

Compile and run **create_holes_delete.c**. Does it work as expected?

18.9 GETTING FILE ATTRIBUTES FROM A FILE INODE

Recall that the UNIX inode contains most of the attributes of a file, including the following:

- Size (in bytes)
- User (Owner) ID
- Group ID (of the owner)
- File mode and access permissions
- Hard link count
- Times for the creation, last access, and last modification
- Pointers to file blocks on secondary storage

Note that the file's name and the current position of the file's read/write pointer are not stored in the inode of the file. The name of the file, as discussed in Chapter 4 and in Section 18.11, is stored in the directory entry for the file and, as discussed Section 18.5.2 the file pointer is stored in the SFT. You can obtain a copy of a file's attributes stored in its inode by using the `stat()`, `lstat()`, and `fstat()` system calls. These calls fill in the stat structure (`struct stat`), defined in **<sys/stat.h>**, with the values of various attributes. We discuss the stat structure and the three calls in Sections 18.9.1 and 18.9.2, respectively.

18.9.1 The stat Structure

Most, but not all, of the attributes of a file are stored in the file's inode. For example, as discussed in Chapter 4, a file's inode number resides in the directory entry of the file and not in file's inode. The stat structure describes part of the inode. The definitions of this structure are similar on both Solaris and FreeBSD. Here is how this structure looks on FreeBSD:

```
struct stat {
 __dev_t  st_dev;               /* inode's device */
 ino_t    st_ino;               /* inode's number */
 mode_t   st_mode;              /* inode protection mode */
 nlink_t  st_nlink;             /* number of hard links */
 uid_t    st_uid;               /* user ID of the file's owner */
 gid_t    st_gid;               /* group ID of the file's group */
 __dev_t  st_rdev;              /* device type */
 struct timespec st_atim;       /* time of last access */
 struct timespec st_mtim;       /* time of last data modification */
 struct timespec st_ctim;       /* time of last file status change */
 off_t    st_size;              /* file size, in bytes */
 blkcnt_t st_blocks;            /* blocks allocated for file */
 blksize_t st_blksize;          /* optimal blocksize for I/O */
 fflags_t  st_flags;            /* user defined flags for file */
 __uint32_t st_gen;             /* file generation number */
 __int32_t st_lspare;
 struct timespec st_birthtim;/* time of file creation */
 unsigned int :(8 / 2) * (16 - (int)sizeof(struct timespec));
 unsigned int :(8 / 2) * (16 - (int)sizeof(struct timespec));
};
```

Most of the fields of the structure are self-explanatory, but not all. Some of the fields that need a bit more explanation are described briefly in Table 18.11.

The **<sys/stat.h>** file defines 24 distinct 24-bit flags for the st _ mode field for file types, access permissions, special permission bits, and so on. A programmer does not need to know what these values are, but you should browse through the header file to see these fields. The **<sys/stat.h>** file also contains several macros to check the type of file. Each macro takes st _ mode value as an argument corresponding to a specified type

TABLE 18.11 Commonly Used Fields of struct stat Defined in **<sys/stat.h>**

Field	**Meaning**
st_mode	• File's access permissions • Type of file • Status of special permission bits: SUID, SGID, and sticky (see Chapter 5)
st_ctim	Time when file's attributes were changed such as file access permissions through the chmod command
st_birthtim	Time when file's inode was created
st_blocks	The number of 512-byte blocks allocated to the file

TABLE 18.12 Macros Defined in **<sys/stat.h>** for Checking the Type of a File

Macro	Test
`S_ISBLK(m)`	Test for block special file
`S_ISCHR(m)`	Test for character special file
`S_ISDIR(m)`	Test for directory
`S_ISFIFO(m)`	Test for named pipe (FIFO)
`S_ISLN(m)`	Test for symbolic link
`S_ISREG(m)`	Test for regular file
`S_ISSOCK(m)`	Test for socket
`S_ISWHT(m)`	Test for whiteout (not implemented)

and evaluates to a non-negative if the test is true. If it evaluates to 0 then the test is false. Table 18.12 contains these macros and their purpose.

18.9.2 Populating the stat Structure with System Calls

You can use any of the three system calls to populate the stat structure with the attributes of a file: stat(), lstat(), and fstat(). Here are brief descriptions of these calls.

```
#include <sys/types.h>
#include <sys/stat/h>
int stat(const char *path, struct stat *sb);
int lstat(const char *path, struct stat *sb);
int fstat(int fd, struct stat *sb);
```

Success: 0
Failure: −1 and kernel variable errno set to indicate the type of error

The stat() system call reads information about the file specified in the path parameter and stores it in the stat structure (struct stat) pointed to by sb. This structure is defined in **<sys/stat.h>**. All of the components in the path variable must be searchable (i.e., must have the execute permission on). Access permissions on the named file are irrelevant because this call does not deal with the contents of the file. If the named file is a symbolic link, the stat() system call returns the information about the file that the link points to.

The lstat() system call works like the stat() system call, except that if the named file is a symbolic link, the call returns information about the link file and not where it points to. The fstat() system call returns information about an open file.

18.9.3 Displaying File Attributes

Once you have obtained file attributes by using one of the preceding system calls and stored them in a variable of struct stat, you can display them by using the Standard I/O functions for output, such as printf() and sprint(). For example, the **stat.c** program that follows reads the sv variable (of struct stat type) the attributes of the file passed as a command line argument. It then displays the size of the file (in bytes) if it is a regular file. Otherwise, it displays an error message using the library call `perror()`.

```
% cat stat.c
#include <sys/stat.h>
#include <sys/types.h>

int main(int argc, char *argv[])
{
       struct stat sv;

      if (argc != 2) {
             printf("Inappropriate number of command line
                    arguments.\n");
             exit(0);
      }
       if (stat(argv[1], &sv) == 0 && S_ISREG(sv.st_mode))
               printf("File size is %d bytes\n", sv.st_size);
       else
               perror("stat");
}

% gcc48 -w stat.c
% ./a.out stat.c
File size is 374 bytes
% ls -l stat.c
-rw-r--r-- 1 sarwar faculty 374 Jan 29 23:52 stat.c
% ./a.out ~
stat: No such file or directory
% ./a.out /usr
stat: No such file or directory
%
```

Note that the error message for the `./a.out ~` and `./a.out /usr` commands is `stat: error 0` on Solaris.

18.9.4 Accessing and Manipulating File Attributes

UNIX provides several system calls for modifying file attributes, including file access permissions, file owner, and file access times. We do not discuss these system calls in detail. Table 18.13 contains brief descriptions of some system calls related to the manipulation of file attributes and their purpose.

EXERCISE 18.14

Browse through the `stat` structure definition on Solaris. What are the differences between the definitions of this structure on FreeBSD and Solaris?

TABLE 18.13 Additional System Calls for Accessing and Changing File Attributes

Accessing and Setting File Attributes	
`access()`	Check file access permissions
`chmod()`	Set file access permissions
`chown()`	Change file owner
`rename()`	Rename a file
`umask()`	Check and set umask
`utime()`	Change access and medication times

EXERCISE 18.15

Compile and run **stat.c**, discussed in Section 18.9.3. Does it work as intended?

EXERCISE 18.16

Browse through the man pages for the system calls given in Table 18.13 on both BSD and Solaris. Write small programs to practice them.

18.10 RESTARTING SYSTEM CALLS

We should write code that has the ability to restart system calls that are interrupted during their execution. Such situations arise when you perform *blocking I/O* using the `read()` and `write()` system calls. We will also discuss blocking I/O in detail in Chapters 19 and 20 while discussing the concept of a communication channel, called a *UNIX pipe*, used for communication between related processes. We discussed the command line use of pipes in detail in Chapter 9.

Simply saying, blocking input means that the `read()` system call waits as there is nothing to read because the pipe is empty. Another example of blocking is when the `read()` system call reads input interactively from a keyboard. The `read()` call blocks until the user enters keyboard input. Similarly, in the context of a pipe, a `write()` system call blocks if the pipe is full.

The `read()` system call may be interrupted when it is performing a blocking read from a slow device. When the `read()` call is used to read input interactively from a keyboard, it may be interrupted while waiting for user input. Most modern UNIX implementations restart such system calls automatically. If you are not sure whether your code would be run on such a system, you need to write code to explicitly handle the restarting of an interrupted system call. The following code snippet may be used for this purpose.

```
Repeat:
       if ((nr = read(fd, buf, SIZE)) == -1)) {
           switch(errno) {
              case EINTR:  goto repeat;
```

```
                        break;
                /* handle other errors */
                default:
            }
        }
```

We address this issue for network-related system calls, particularly, the `select()` system call, in Chapter 20.

18.11 SYSTEM CALLS FOR MANIPULATING DIRECTORIES

UNIX provides a set of system calls for creating, deleting, and changing directories. Table 18.14 lists these calls along with their purpose. We will not discuss these calls any further. However, you will get to practice them in a programming exercise given at the end of the chapter.

As a system programmer, you do not need to know the details of the kernel data structures for directories. However, such knowledge enhances your understanding of the UNIX operating system and your skills as a programmer. The most important directory data structure is defined in the **<sys/dirent.h>** file. Here is the definition of the directory entry in Solaris that is independent of the file system.

```
struct dirent {
   ino_t           d_ino;     /* "inode number" of entry */
   off_t           d_off;     /* offset of disk directory entry */
   unsigned short  d_reclen;  /* length of this record */
   char            d_name[1]; /* name of file */
};
```

In FreeBSD, it the directory entry is defined as

```
struct dirent {
  __uint32_t d_fileno;      /* file number of entry */
  __uint16_t d_reclen;      /* length of this record */
  __uint8_t  d_type;        /* file type, see below */
```

TABLE 18.14 UNIX System Calls Related to Directories

System Call	Purpose
`chdir()`	Change directory
`mkdir()`	Create directory
`rmdir()`	Remove directory
`remove()`	Remove an empty directory; equivalent to `rmdir()`
`rename()`	Rename a directory

```
  __uint8_t  d_namlen;       /* length of string in d_name */
  char    d_name[255 + 1];  /* name must be no longer than this */
};
```

Definitions in both systems have three common fields: the inode number for the file (called the file number under FreeBSD), the file name, and the length of the directory entry. The BSD definition contains two additional fields: the length of the string in the name field and the type of the file the entry represents. The type field is not strictly required here because the inode contains this information, as discussed in Section 18.9. The Solaris definition contains the additional field of offset of the disk directory entry.

You can use several C library functions to manipulate directories, including `opendir()`, `closedir()`, `readdir()`, `telldir()`, `seekdir()`, and `rewinddir()`.

EXERCISE 18.17

Browse through the man pages for the previously stated calls to manipulate directories in order to understand their syntax and semantics. Then write small programs to use them in order to enhance your understanding of these calls.

18.12 IMPORTANT WEB RESOURCES

Table 18.15 lists useful websites for UNIX system programming and related topics.

TABLE 18.15 Web Resources for the UNIX System Programming and Related Topics

`https://www.freebsd.org/`	Home page for FreeBSD. Contains a lot of useful material, including FreeBSD source, manual pages, support, SVN repository, forums, user groups, etc.
`http://www.oracle.com/technetwork/server-storage/solaris11/documentation/index.html`	Webpage for Oracle Solaris 11 product documentation.
`http://www.gnu.org/`	Homepage for the free operating system GNU by the Free Software Foundation. Work on GNU's kernel, The Hurd, started in 1990 (before work on Linux had started).
`http://www.tutorialspoint.com/index.htm`	An excellent site for tutorials on almost all known programming languages, including C, and development environments.
`http://www.tutorialspoint.com/c_standard_library/index.htm`	The C standard library.
`www.compileonline.com (http://www.tutorialspoint.com/codingground.htm)`	A great free site for online editing, compiling, and running programs in almost any language from assembly language to Python and from Algol-68 to Ruby.

SUMMARY

UNIX provides two APIs for software development: the language libraries and the system call interface. Application programmers typically use the interface provided by the language libraries and system programmers primarily use the system call interface. We described the UNIX system calls and their use in small programs for the I/O of file data and attributes. Our focus was on regular files.

Before discussing the system calls, we explained that four types of events cause the control to be transferred to the kernel code. They are interrupt, trap, signal, and system call, with the first two caused by the system hardware and the last two primarily caused by software. We also discussed the details of events that happen in order to execute a system call. We discussed briefly the C Standard Library and the difference between Standard I/O and low-level I/O through system calls.

We discussed the system calls for data I/O for regular files and I/O of file attributes using the `open()`, `creat()`, `read()`, `write()`, `close()`, `truncate()`, and `stat()` system calls. For random I/O, we used the `lseek()` system call to set the file pointer to the byte where the next read or write operation should be performed. We showed the use of the `lseek()` and `write()` system calls to illustrate how holes can be created in a regular file under UNIX. We showed the holes created in a file by displaying its octal dump using the `od` command.

We discussed briefly several system calls and C library functions for accessing and manipulating file attributes and directories. These calls are: `access()`, `chdir()`, `chmod()`, `chown()`, `closedir()`, `mkdir()`, `opendir()`, `read()`, `readdir()`, `rename()`, `rewinddir()`, `seekdir()`, `telldir()`, `umask()`, `utime()`.

We also showed the kernel data structures for the `stat` structure and a directory entry, as described in the **<sys/stat.h>**, **<limits.h>**, and **<sys/limits.h>** files.

Finally, we showed how you can use a small piece of code to restart a system call if it is interrupted in the middle of execution and the underlying OS kernel does not automatically restart it. Such interruption may occur when blocking I/O is done on slow devices, keyboards, and files such as pipes, FIFOs, and sockets.

QUESTIONS AND PROBLEMS

1. What is a system call? List names of three system calls in each of the following categories:

    a. Process control

    b. File management

    c. Device management

    d. Information maintenance

    e. Communications

2. Clearly describe the difference between file I/O using the Standard C library and the UNIX system call interface.
3. What is a stream in the context of Standard I/O? Explain with an example.
4. What is the `FILE` object? What does it contain? Create a file called **foo**, open it using a Standard C library function and display the file descriptor of the file.
5. Describe clearly the differences between the PPFDTs and the SFT. Illustrate your answer with an example.
6. Describe in words situations under which the same entry in multiple PPFDTs would point to the same entry in the SFT.
7. What happens when a process executes the `close()` system call on a file descriptor when pointers from more than one PPFDT point to the entry for this file in the SFT?
8. What is the size of the PPFDT on Solaris and BSD? *Hint*: Browse through the <**limits.h**> file on Solaris and <**limits.h**> and <**sys/syslimits.h**> files on BSD.
9. What does the `lseek()` system call return after completing successful execution?
10. Write a piece of code that opens a file whose name is passed as a command line argument and sets the read/write file pointer to the beginning of the file.
11. Write a program that takes the following command line arguments: `file`, `position`, `nbytes`. It displays `nbytes` bytes from the file starting with the `position` byte of the file. Do the appropriate error handling.
12. Enhance the **create_holes.c** program in Section 18.8.5 so that it takes the hole size from the command line and appends after the hole data taken from the file specified as the second command line argument. Compile and show a few sample runs of the program.
13. What is the effect of executing the `unlink()` system call if there are multiple hard links to a file?
14. Give the syntax for the `open()` system call for creating a new file for reading and writing.

    ```
    open(pathname, O_RDWR | O_CREAT | O_TRUNC, mode);
    ```
15. Give the sequence of system calls for performing this operation by using the `creat()`, `open()`, and `close()` system calls.
16. Write a C program to verify that the UNIX kernel allocates file descriptors sequentially starting with descriptor 0. Show the source code and execution of your program.
17. Create a large file running the `man csh >> bigdata` command five times. Then, create three holes in it of sizes 4K, 8K, and 16K bytes. Display the file size by using the `ls -l bigdata` command. Then use the `du bigdata` command to determine the

file size. Read the man page for the du command to determine the block size used so you can perform this task. Now verify that the **bigdata** file contains holes equal to 28K bytes.

18. What is the purpose of the following program? Explain your answer.

```
int main(void)
{
   int nr, nw;
   while ((nr = read(0, &c, 1)) !== 0)
       nw = write(1, &c, 1);
   return(0);
}
```

19. Write a C program that takes a file as a command line argument and outputs Yes if the file is empty and No if it isn't. Show the source code, compilation of the source code, and a few sample runs of the program.

20. What would happen if we did not use the O _ TRUNC flag in the **cpy.c** program in Section 18.8.4?

21. Modify the **truncate.c** program discussed in Section 18.8.6 so that it takes any number of files as command line arguments and truncates them to the argument specified as the last argument. It displays the names and sizes of the original and truncated files in the form of a table and removes the truncated files from the file system. Show a few sample runs of the program.

22. Change the **truncate.c** program discussed in Section 18.8.6 so that it takes three command line arguments. The first two arguments are file names, with the first the name of an existing file. The third argument specifies the new file length. The program creates the file with the name given as the second argument, copies the first file into it, truncates the file according to the third argument, displays the sizes of the original and truncated file, removes the truncated file from the file system, and terminates. Show a few sample runs of the program.

23. Browse through the **<stdio.h>** file and identify the values (or macros) for the following:

    - Maximum number of files that the system allows to be opened concurrently
    - Macros for standard files
    - Macros that define SEEK _ CUR, SEEK _ END, and SEEK _ SET

24. Browse through the **<stdio.h>** file and identify all of the functions and their purposes.

25. Write a program that opens a file with one hard link and removes the file with the unlink() system call. Read data from the opened file and send it to standard output using the write() system call. What happens? Is your program able to read file data? Explain your answer.

26. Clearly state the difference between the `stat()`, `fstat()`, and `lstat()` system calls. Which of these calls would you use if you need to display the attributes of a link-type file? Why?

27. Write a C program that uses the `stat()` system call to display the file size and last modification time for the file whose path is passed as a command line argument. The program then opens the file, appends data to it, and uses the `fstat()` system call to read and display the file size and modification time for the file.

28. Enhance the program from Problem 27 so that it takes the file as a command line argument and, in the output, first displays the type of the file followed by the file size in bytes, the file's inode number, the file's access permissions, the number of hard links to the file, the file size in 512-byte blocks, and the last modification time. Compile the program and show its execution for an ordinary file, a directory, a link file, a character special file, and a block special file.

29. Modify the **stat.c** program discussed in Section 18.9.3 so that if the command line argument is a link file, it displays the information about the link file and not where the link points. Do the appropriate error handling.

30. Write a simple UNIX shell that:

    a. Displays a prompt and waits for the user to enter a command terminated with `<Enter>`

    b. Executes the shell commands and prompts the user for input again

    c. Terminates when the user presses `<Ctrl+D>`

    You should implement the following commands with your own code; that is, they are the built-in (intrinsic) shell commands:

Command	Purpose
`finfo file`	Display the following attributes of file: type, number of hard links, user ID, access permissions
`truncate file size`	Truncate size of file to `size` bytes
`holes file number`	Create `number` holes in file if it is a regular file
`mkdir dname`	Create the directory dname
`rmdir dname`	Remove the directory dname
`cd [dname]`	Change directory to dname

    For the execution of all other shell commands, use the library function `system()`. Do the appropriate error handling. You will need to use the following system and library calls in addition to the Standard I/O library calls: `chdir()`, `mkdir()`, `rmdir()`, `truncate()`, `lseek()`, `stat()`, and `system()`.

CHAPTER 19

System Programming II

Process Management and Signal Processing

Objectives

- To explain the concepts of processes and threads
- To describe the general concept of the process control block (PCB) and its implementation in UNIX
- To discuss the details of main memory and disk images of UNIX processes
- To describe threads in detail, including user- and kernel-level threads
- To explain the differences and commonalities between threads
- To discuss fundamental process management concepts in UNIX: process creation, process termination, waiting for a child process to terminate and obtain its termination status, overwriting a process image with another, creating zombie processes, sharing open files between a process and its children, and duplicating file descriptors
- To explain the concept of signals in UNIX, signal generation, and signal handling
- To explain process management concepts using various system and library calls within several small programs
- To explain briefly several system calls and C library functions for manipulating directories and file attributes
- To cover the system calls, library calls, commands, and primitives

  `alarm(), dup(), dup2(), execl(), execve(), execle(), exit(), file, fork(), gcc48, getpid(), getppid(), getuid(), getgid(),` `kill()`, `ls`, `open()`, `signal()`, `size`, `wait()`, `waitpid()`, `wait3()`, `wait4()`, `wait6()`

19.1 INTRODUCTION

In this chapter, we discuss the use of the API provided by the UNIX SCI and libraries for process creation, termination, and management. We also describe the generation and handling of software interrupts, commonly known as *signals* in UNIX lexicon. Our coverage starts with the concepts of processes and threads, and moves to the coverage of the UNIX process control block, the structure of the memory and disk images of UNIX processes, static and dynamic linking, single- and multithreaded processes, user- and kernel-level threads, and the differences between processes and threads. We also touch on the issues of the *critical section* and *critical section problems*.

Our treatment of UNIX process management encompasses the following: system and library calls for process creation, process termination and status reporting to the parent process, getting the process ID and parent process ID, creating and handling zombie processes, a process overwriting itself with another executable image, duplicating file descriptors, file sharing between processes, sending different types of interrupts (UNIX signals) to processes, handling signals, and setting up alarms in programs. As usual, our coverage of the various topics is closely linked with the underlying kernel data structures and operating system concepts where appropriate. As was the case for the programs in Chapter 18, for the compilation of our sample C programs we use the default GNU C compilers on PC-BSD and Solaris. The default compiler on our PC-BSD and Solaris systems are `gcc48` and `gcc`, respectively. We show the compilation and execution of our programs on PC-BSD.

19.2 PROCESSES AND THREADS

In this section, we discuss the concepts of processes and threads. Our discussion focuses on the conceptualization of the two, how they are created, the system resources that they need for their execution, their address spaces, similarities between two, and how they differ from each other. After the discussion on generic processes, we focus primarily on UNIX processes. The discussion of threads covers both user- and kernel-level thread libraries.

19.2.1 What Is a Process?

A simplistic and high-level view of a process is that it is a program in execution. Process execution starts at the entry point into the process (usually the first statement of the main function) and continues sequentially, one program statement at a time. Thus, a program counter is associated with a process that controls the sequence of execution of the program statements, including function calls. On a function call, the control transfers to the called function and the first statement in the function is executed, followed by the execution of the remaining body of code in the function sequentially, statement by statement. On return from the function, the statement following the function call in the caller function is executed. Process execution continues like this until the execution of the last statement in the process.

A deeper look into a process reveals that it in fact consists of the following three entities:

1. The address space
2. The CPU state
3. The process control block

The *address space* of the process is dictated by the main *memory image* of the process, as discussed in Chapter 10, Section 10.5.1. The values of the CPU registers at any given time, including the value of the program counter, comprise the *CPU state* of the process. We describe the enhanced version of the main memory image of a process, the address space of a process, and the *process control block* (PCB) of a process in the next subsection.

19.2.2 Process Control Block

The PCB is a kernel data structure that keeps track of the run-time attributes of the process. In UNIX, the PCB of a process consists of two parts: the *proc structure* and the *u area*. Whereas the proc structure contains the scheduling-related information of a process, the u area contains information about signal handling, resource allocation, and a reference to the proc structure. The proc structure of a process always remains in the main memory regardless of the state of the process. However, the u area is in the main memory only when the process is in the running state.

The proc structure contains the PID and PPID, its priority, scheduling and waiting queues, information about memory management (paging and segmentation), the process state, and the signal-handling mask. The u area contains a pointer to the proc structure, the CPU state (i.e., context) of a blocked process, the UID and GUID, the current directory, CPU usage of the process, the terminal the process is attached to, signal-handling information, and the PPFDT.

A process may not access its PCB directly; the kernel updates the proc structure fields of a process as and when needed. For example, the kernel code updates the scheduling priority field in the PCB of a process when the process priority changes. Although a part of a process image, the u area is only accessible to a process through the kernel code that executes on behalf of the process, such as the code for a system call.

19.2.3 Process Memory Image (Process Address Space)

We discussed the memory image of a UNIX process in detail in Chapter 10. As stated earlier, the process memory image, also known as the *process image*, consists of several sections, as shown in Figure 10.6. Figure 19.1 shows the complete version of the process image, including the u area at the top of the image. The memory image of a process delineates the main memory region(s) that a process may access legally, known as the *process address space*.

Some of the regions of the process image come into being only while the process remains in the system (running or waiting for an event); other regions are an integral part of the process image, whether it is a process in the main memory or an executable image on

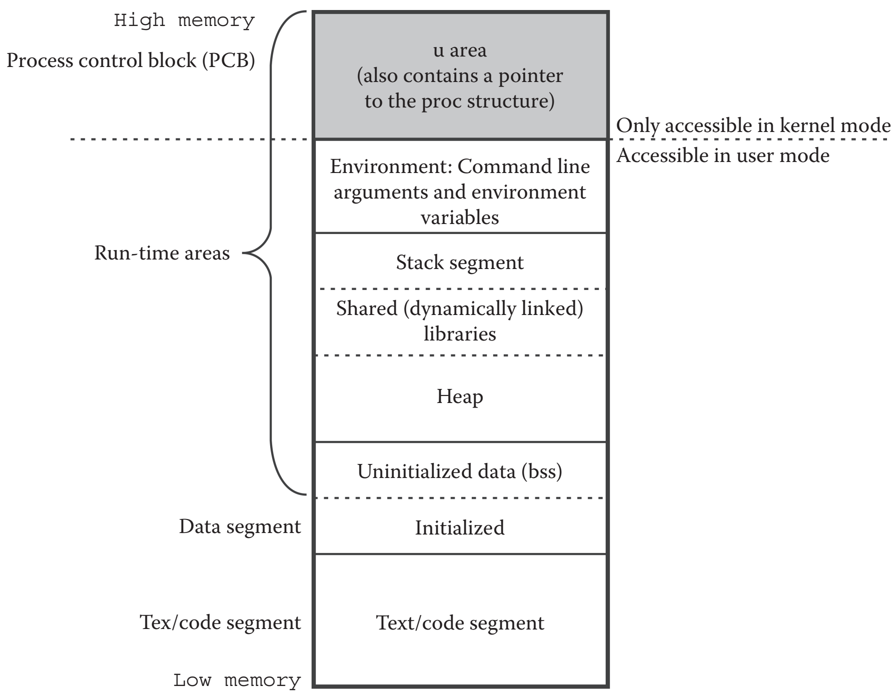


FIGURE 19.1 Main memory image (process address space) of a UNIX process.

disk. For example, the environment, stack, shared (dynamically linked) libraries, heap, and uninitialized data portions of the process are only required for as long as the process remains in the system. These regions have been labeled as "Run-time areas" in Figure 19.1. We have discussed most of the sections of the address space of a process in this and/or other chapters, except shared libraries.

By default, all modern C compilers, including gcc, generate executable code by linking all library calls to the relevant library code at run time. When a process calls a library function, the code for the relevant library is loaded into the memory if it is not already in the memory because of a previous reference to this library by this or another process. Once a library's code has been loaded into the memory, it remains memory resident for future references to it by any process. Such linking of library code to an executable code is called *dynamic linking* and the libraries used in this fashion are called *shared libraries*. Thus, dynamically linked libraries are always shared.

Dynamic linking is preferred over static linking, primarily, for the following reasons:

1. The resultant executable code is smaller in size. It means that it requires less disk space to save it, a shorter time to load it into the main memory, and potentially less main memory to execute it (in case the library function is not called during execution).

2. Library code is loaded into the memory only once and is shared by multiple processes.

3. New executable code does not need to be generated (relinked) again if a library is updated, provided the prototypes for the library calls do not change.

The primary drawback of dynamic linking is that the execution of a program is slower if it refers to a library that has not been referenced by any process previously and, hence, its code has not been loaded into the main memory previously.

19.2.4 Process Disk Image

The disk image of an executable file in UNIX has five sections, as shown in Figure 19.2. These sections are: header, text/code, data, relocation information, and symbol table. The header section contains the following information:

- Magic number
- Size of text/code area
- Size of data area
- Size of initialized data area (bss)
- Size of the symbol table
- Information about the entry point into the text/code area and flags

The *magic number* of an executable file describes the type of the executable code in the file. A few types are: binary executable generated by a compiler, shell script, Perl script, and Python code. Although commonly used in the computer literature now to describe file formats and protocols, the term *magic number* was first used in the seventh edition of UNIX for identifying the type of an executable file. The UNIX command `file`, discussed in Chapter 4, uses the magic number in a file to decipher the type of the file.

The relocation information describes whether the program is *relocatable*. An executable code is relocatable if it will run regardless of where it is loaded into the memory. Programs

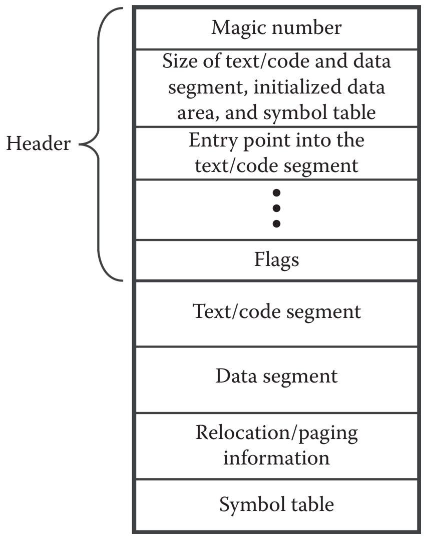

Header
Magic number
Size of text/code and data segment, initialized data area, and symbol table
Entry point into the text/code segment
Flags
Text/code segment
Data segment
Relocation/paging information
Symbol table

FIGURE 19.2 Disk image of an executable file.

that are not relocatable must be loaded in a specific area in the main memory for it to execute properly.

As discussed in Chapter 10, you can use the `size` command to display the sizes of the text/code, data, and bss segments of an executable file. The following session shows the use of the `file`, `ls –l`, and `size` commands to display the types of the executable files **a.out** and **cpy**, their sizes in bytes, and the sizes in bytes of their code, data, and bss segments, as well as the sum of their sizes in bytes shown in decimal and hexadecimal notations.

```
% file a.out cpy
a.out: ELF 64-bit LSB executable, x86-64, version 1 (FreeBSD),
dynamically linked (uses shared libs), for FreeBSD 10.0 (1000510),
not stripped
cpy:   ELF 64-bit LSB executable, x86-64, version 1 (FreeBSD),
dynamically linked (uses shared libs), for FreeBSD 10.0 (1000510),
not stripped
% ls -l a.out cpy
-rwxr-xr-x  1 sarwar  sarwar  8153 Feb  7 22:52 a.out
-rwxr-xr-x  1 sarwar  sarwar  7960 Feb 21 14:32 cpy
% size a.out cpy
   text   data    bss    dec    hex filename
   2599    520     24   3143    c47 a.out
   2484    504     24   3012    bc4 cpy
%
```

The output of the `file` command shows that **a.out** and **cpy** are both executable files in the ELF 64-bit format. The output of the `ls –l` command shows that their sizes are 8153 and 7960 bytes, respectively. Finally, the output of the `size` command shows the sizes in bytes of their text, data, and bss segments, as well as the sum of their sizes.

EXERCISE 19.1

What is the difference between static linking and dynamic linking?

EXERCISE 19.2

Use the `size` command to determine the sizes of the text, data, and bss segments of the executable files for the UNIX commands `find` and `sort`. Show the command runs and their outputs.

19.2.5 What Is a Thread?

When a process calls a function from its "main" function, the control of execution transfers from the main function to the code of the *called function*. On return from the function, the statement following the function call is executed in the *caller function*. If a called function calls another function, the control transfers to the newly called function, returns to its caller, and, eventually, returns to the main function. Thus, there is a single flow of control as the program executes, also known as the *thread of execution* (commonly called a *thread*), that, usually, starts with the main function and moves to the called functions one after another, eventually returning to the main function. When the main function finishes its execution, the only thread of execution in the process and the process itself terminate. Such processes are known as *single-threaded processes* and the only thread in them is known as the *main thread*. We can also say that single-threaded processes have only one *program counter* (PC) associated with them and the value of the PC at any given time determines the address of the next instruction to be executed in the thread.

All conventional programs result in single-threaded processes. Figure 19.3 shows a typical single-threaded C program in which program execution starts with the main function and moves through different functions when they are called. The single execution path followed by the process from `main` to other functions and back—that is, the thread of execution through the program—is shown in Figure 19.4. Note how the only thread of execution moves from one function to another and back in the following order:

main → f1 → f2 → f1 → main → f3 → main → f1 → main

Code

```
#include <stdio.h>

int main(int argc, char* argv[])
{
         int i;
         char c;

         printf(“Hello, world!\n”);
         ...
         f1(...);
         ...
         f3(...);
         ...
         f1(...);
         ...
         exit(0);
}
void f1(int i, int* ip)
{
         ...
         if (...)
         then
                f2(...);
         else
                ...
}
void f2(char c)
{
         ...
}
void f3(char* s)
{
         ...
}
```

FIGURE 19.3 Single-threaded process with only the main thread.

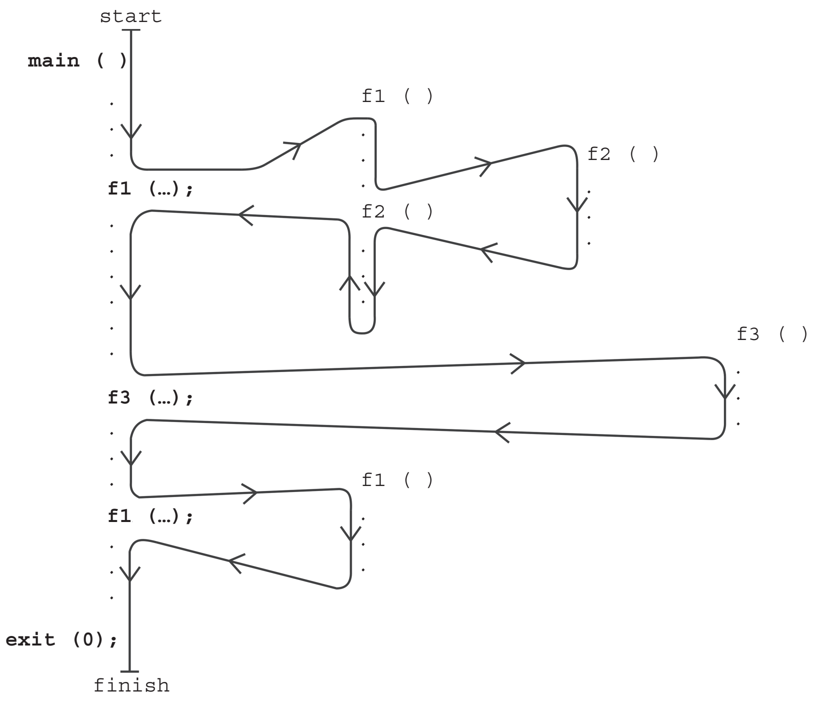


FIGURE 19.4 The flow of control (thread) of the program: a single-threaded process.

You can create another thread of execution in your program that executes in parallel with the main thread by executing a piece of code in the program, usually a function, such that the function code executes like an independent program but without a main function and within the address space of the process in which the function resides. This means that this thread of execution has its own program counter and stack. However, because it executes within the address space of a process, it shares with all other threads of execution, including the main thread, data, code, and other segments of the program previously discussed.

You may use the API for one of several thread libraries to create and manage threads. The threads created by user programs using these libraries, which are not known to the kernel and are managed solely by the user-level thread libraries, are known as *user-level threads*. The code and data structures for these libraries are maintained in the user space. The kernel does not know about and provides no support for such threads. It means that the kernel only manages processes and the relevant thread library manages threads, including their scheduling.

Some programming languages such as Java provide direct support for user-level threads. Such threads are created and managed by the program itself through the use of the API of a thread library.

If a thread library is implemented in the kernel, the operating system maintains the code and data structures for the library. Calls to functions in these libraries for creating and managing threads result in system calls. The threads created by the user programs using such libraries are known as *kernel-level threads*. The kernel handles both processes and threads, including their scheduling.

Several libraries are available for the implementation of user- and kernel-level threads. The POSIX standard *Pthreads* may be provided as user- or kernel-level libraries. Win32 threads are available as kernel-level libraries.

Kernels of almost all modern general-purpose operating systems are multithreaded. This means that they can serve multiple system calls simultaneously. This is done by running the code corresponding to a system call as a thread instead of a function call. Thus, for example, multiple invocations of the `read()` system call from threads within the same process or different processes may be served simultaneously. The operating system kernels that offer such multithreaded services are known as *multithreaded kernels*.

We use library calls to create, terminate, and manage threads. If the threads library were for kernel threads, each library call would eventually invoke a system call. The kernel would have the knowledge of threads and would be responsible for managing them, including their scheduling. Otherwise, the user-level library manages threads.

In Figure 19.5, we show a multithreaded process with four threads: the main thread, two threads of the function `f1()`, and one thread for the function `f3()`. A function that is designed to become a thread must terminate with a library function that terminates the thread, such as `pthread _ exit(...)`. We use the names of the Pthreads library functions for the creation and termination of threads. Note that you need to include the header file **/usr/include/pthread.h** in your program and link the Pthreads library with the executable code using the `–pthread` option with the gcc compiler, as in

```
gcc48 –w –pthread prog.c –o prog
```

The typical graphical representations of single-threaded and multithreaded processes are shown in Figure 19.6.

19.2.6 Commonalities and Differences between Processes and Threads

Processes and threads have several things in common. For example, each has its own ID, stack, and program counter.

There are several differences between processes and threads. For example, whereas processes operate within their own address spaces, threads within a process operate within the address space of the process. Similarly, processes have their own data and text (code) areas but threads share the text and data areas in the process.

19.2.7 Data Sharing among Threads and the Critical Section Problem

Data sharing among threads is a mixed blessing. Whereas this saves memory space by preventing the duplication of data, it does require that threads access data on a mutually exclusive basis by locking data before accessing it and unlocking it afterward. If this were not done, the results produced by the process would be dependent on the sequence in which instructions within threads access shared data, and the results may or may not be correct. This is called the *race condition* and the final result produced by the process is dependent on the order in which instructions in different threads access the shared data.

Code

```
#include <stdio.h>
#incldue <pthreads.h>

int main(int argc, char* argv[])
{
        int i;
        char c;

        printf("Hello, world!\n");
        ...
        pthread_create(..., ..., f1, ...);
        ...
        pthread_create(..., ..., f3, ...);
        ...
        pthread_create(..., ..., f1, ...);
        ...
        exit(0);
}
void f1(int i, int* ip)
{
        ...
        if (...)
        then
                f2(...);
        else
                ...
         pthread_exit(...);
}
void f2(char c)
{
        ...
         pthread_exit(NULL);
}
void f3(char* s)
{
        ...
         pthread_exit(NULL);
}
```

FIGURE 19.5 Multithreaded process with four threads including the main thread.

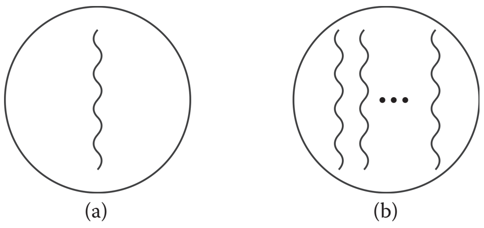


FIGURE 19.6 Typical graphical representations of (a) a single-threaded process and (b) a multithreaded process.

The piece of code in a thread that accesses shared data is known as the *critical section*. Thus multiple critical sections that access shared data must be executed mutually exclusively, one after the other. In other words, when there are simultaneous requests for the execution of multiple critical sections, the execution of these critical sections must be serialized. How such mutual exclusion is achieved is known as the *critical section problem*. Several solutions are available in the literature for solving this problem. However, further discussion on this topic is beyond the scope of this book. If you are interested in knowing more about this subject, you may read a textbook on operating system principles or operating system concepts.

EXERCISE 19.3

Browse the Web and write down the names of three thread libraries, in addition to those we have discussed in Section 19.2.5, that may be used for user-level threads.

EXERCISE 19.4

The Bakery Algorithm is an elegant solution for the N-process critical section problem. Browse the Web or see a book on operating system principles and concepts to find out the name of the author of this algorithm. Who is the author?

19.3 PROCESS MANAGEMENT CONCEPTS

Process management entails several things, from process creation to termination and everything in between, including suspending a process, having a process wait for an event such as the termination of a child process, sending a signal (software interrupt) to a process, handling signals, setting up an alarm in a process, and duplicating a file descriptor in the PPFDT. Table 19.1 shows a few system calls for process management.

TABLE 19.1 The UNIX System Calls for Process Management

`fork()`	Create a clone (i.e., a child) of the calling process
`execve()`	Overwrite the main memory image of the caller process
`wait()`	Suspend the calling process and wait for a child process to terminate
`waitpid()`	Wait for the termination of the process with the given PID
`exit()`	Terminate the caller process and return status to the parent of the process
`getpid()`	Get the PID of the caller process
`getppid()`	Get the PPID of the parent of the caller process
`getuid()`	Get the UID of the owner of the caller process
`getgid()`	Get the GID of the owner of the caller process
`signal()`	Handle a signal by ignoring it, taking the default (kernel-defined) action, or taking the programmer-defined action specified for the relevant signal
`kill()`	Send a signal of a particular type to a process; normally used to terminate a process
`alarm()`	Set an alarm signal for the given number of seconds
`dup(), dup2()`	Duplicate a file descriptor in the PPFDT

EXERCISE 19.5

Browse the man page for the `execve()` system call and list the names of all the variants of this call, both system calls and library calls.

EXERCISE 19.6

Browse the man page for the `wait()` system call and list the names of all the variants of this call, both system calls and library calls.

19.3.1 Getting the Process ID and the Parent Process ID

You can use the `getpid()` and `getppid()` calls to display the ID of a process (PID) and the ID of its parent (PPID). Here are the brief descriptions of these calls.

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid();
```

Success: The ID of the process (PID)
Failure: –1 and kernel variable `errno` set to indicate the type of error

```
#include <sys/types.h>
#include <unistd.h>
pid_t getppid();
```

Success: The ID of the parent process (PPID)
Failure: –1 and kernel variable `errno` set to indicate the type of error

The **pids.c** program in the following session shows the use of these calls. The compilation and running of this program displays the PID of the process (26438) and the PID of its parent (16733). Note that the parent of the process is the login shell (`-csh`) with PID 16733, as shown in the output of the `ps` command.

```
% cat pids.c
void main(void)
{
       printf("Child's PID = %d\n", getpid());
       printf("Parent's PID = %d\n", getppid());
}
% gcc48 -w pids.c -o pids
% ./pids
Child's PID = 26438
Parent's PID = 16733
% ps
  PID TT  STAT    TIME COMMAND
16733  1  Ss   0:02.23 -csh (csh)
26830  1  R+   0:00.01 ps
%
```

EXERCISE 19.7

Compile and run the **pids.c** program to make sure it works on your system.

EXERCISE 19.8

Browse the man page for the `getpgrp()` and `getpgid()` system calls. What is the purpose of each of these calls?

19.3.2 Creating a Clone of a Process

The only way to create a process in UNIX is to have a process create a clone of itself—that is, a replica of the main memory image of the process and most of the associated kernel data structures (discussed later in this section). A process can use the `fork()` system call for this purpose. The clone is called the *child process* and the caller of `fork()` is known as the *parent process*. Both processes have their unique process IDs (PIDs). Here is a brief description of the `fork()` system call.

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```
Success: 0 to the child process and the PID of child to the parent process
Failure: –1 and kernel variable `errno` set to indicate the type of error

As you can see, the `fork()` call returns a value of type `pid _ t`, which is defined in the header file **/usr/incude/sys/types.h** as a 32-bit integer. This means that you would use `fork()` in an assignment statement, as in

```
pid_t pid;
...
pid = fork();
...
```

As soon as `fork()` has completed the creation of the child process and before it returns, the parent and child processes start running concurrently as independent processes. Both execute the statement following `fork()`, which is the assignment statement meant to save the return value of the `fork()` call. The `fork()` call returns 0 to the child process and the PID of the child process to the parent process. It is not easy to comprehend how a system call could return two values because a function may return only one value. Note that it is possible for `fork()` to return two values because as soon as the child process has been created `fork()` has to complete its execution in both processes by returning a value in each of the two processes.

The child process inherits several attributes and characteristics of the parent process. Here is a partial list:

- A copy of the parent's PPFDT
- The current working directory

- The environment
- The root directory
- The set-user-ID (SUID) and set-group-ID (SGID) status
- The signal settings
- The time left before an alarm goes off
- The value of the file creation mask, umask

Although the child process is an exact copy of the memory image of the parent process, the two differ from each other in many ways:

- Both processes have their own process IDs (PIDs).
- Both have different parent processes.
- Both have their own copies of the PPFDT.
- The resource utilization for the child process is set to 0.
- The child process has only one thread of execution, and if the parent process is multithreaded, the other threads do not release the resources held by them.

Since the parent and child processes have copies of the same PPFDT, both can do I/O with files that the parent process had opened before creating the child process and this I/O is visible in both processes. In other words, the change is the position of the file pointer due to `read()`, `write()`, and `lseek()` calls by either process is visible in the other process. So, if either process reads N bytes from an already open file, the file pointer is incremented by N and the next read or write operation by parent or child is performed at the new position of the file pointer.

The `fork()` call may fail for the reasons listed in Table 19.2.

The `vfork()` system call is a lighter version of `fork()`. It creates a child process exactly like `fork()` but does not copy the address space of the parent; it copies the address space on demand—that is, when the child's memory image is needed. The `vfork()` system call is used instead of `fork()`, particularly if the child process uses the `execve()` system call (or a variant) or the `execl()` library call (or a variant) right after its creation to overwrite itself

TABLE 19.2 Reasons for the `fork()` System Call to Fail

Reason for Failure	**Value of** `errno`
The limit on the maximum number of processes that may run on the system simultaneously would be exceeded	EAGAIN
The limit on the maximum number of processes that may run under a user would be exceeded	EAGAIN
The resource limit set for the process has been reached	EAGAIN
Insufficient swap space for the new process	ENOMEM

with another executable code. This is done in applications like a shell program that executes external commands by using and `exec()` call (`execl()`, `execve()`, `fexecve()`, etc.) immediately after `fork()`. The `exec()` call is covered later in the chapter.

EXERCISE 19.9

If you browse the man page for the `fork()` system call, you will see the names of the `vfork()` and `rfork()` calls. What is the purpose of each of these calls?

19.3.3 Reporting Status to the Parent Process

A UNIX process reports its termination status to its parent via the `_exit()`system call or the `exit()` library call. Since most programmers use the `exit()` call to terminate a process, we will discuss it throughout the book. A process passes its termination status to its parent through the only parameter of the `exit()` call, usually, 0 or 1. The value 0 is used to indicate successful completion of the process and 1 means its unsuccessful termination.

The parent process accepts this status through the use of a system call in the `wait()` class (see Section 19.3.4). In reality, the status of the terminating process is saved in the proc structure of the process. All the data structures and resources allocated to the process are deallocated, except its proc structure. The proc structure is released back to the kernel after the parent has read the exit status of the child process. If a process does not use the `exit()` call to terminate itself, the kernel automatically sends the termination status of the process to its parent process.

Note that a child process and its parent rendezvous through the `exit()` and `wait()` system calls, with the parent receiving the exit status of the child and the reason for its termination through this meeting. This is the simplest form of interprocess communication available in UNIX. By associating different connotations to the `exit()`parameter, the child can communicate meaningful messages to its parent. We discuss the issue of communication between UNIX processes in detail in Chapter 20.

19.3.4 Collecting the Status of a Child Process

A process can use the `wait()` system call to wait for the termination of a child process and to know the reason of its termination. The process can use the `waitpid()` system call to wait for the termination of a particular child process. Here are brief descriptions of these system calls.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t wpid, int *status, int options);
```

Success: PID of the child process
Failure: –1 and kernel variable `errno` set to indicate the type of error

The `wait()` call may fail for several reasons. Some of these reasons are listed in Table 19.3.

TABLE 19.3 Some of the Reasons for the `wait()` System Call to Fail

Reason for Failure	Value of `errno`
The caller process has no children to wait for	`ECHILD`
The statuses from the terminated children are not available because the caller process is ignoring `SIGCHLD` and the system is discarding such statuses	`ECHILD`
The call was interrupted by a signal	`EINTR`
An invalid option was specified in the system call	`EINVAL`

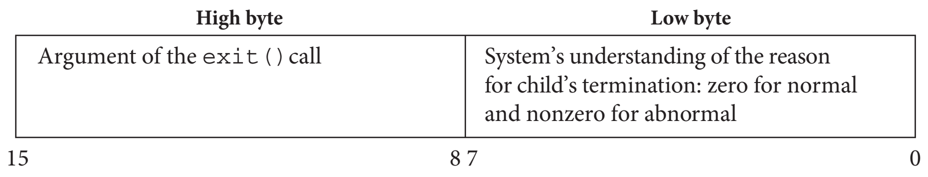

High byte
Low byte
Argument of the exit() call
System's understanding of the reason for child's termination: zero for normal and nonzero for abnormal
15
8 7
0

FIGURE 19.7 The meaning of the values in the two bytes of the value in the `status` parameter.

The programmer can decipher the termination status of a child process and additional information related to its termination, such as whether the process was terminated due to a signal, by examining the value of the `status` parameter. As shown in Figure 19.7, the exit status and the reason for the process to terminate are stored in `status`. The lower byte contains the reason for the process termination and the upper byte contains the exit status. If a process terminated on receiving a signal, the signal number is stored in the lower byte of the status variable. Thus, for example, if a process terminated due to `<Ctrl+C>` (keyboard interrupt), the lower byte contains `SIGINT` (i.e., value 2).

With the passage of time, UNIX designers have added a number of variants of the `wait()` call. The `wait()` system call is the oldest and `waitpid()` is the most restricted of these calls. The newest and broadest interface is the `wait6()` system call. Detailed discussions on `wait3()`, `wait4()`, and `wait6()`are beyond the scope of this book. Here are brief descriptions of the `wait3()` and `wait4()` calls.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>
pid_t wait3(int *status, int options, struct rusage *rusage);
pid_t wait4(pid_t wpid, int *status, int options, struct rusage *rusage);
```
Success: PID of the child process
Failure: –1 and kernel variable `errno` set to indicate the type of error

The `wait4()` system call instructs the process to wait for specific children processes and to retrieve the statistics about their resource usage. As shown in Table 19.4, the `wpid` parameter in the `wait4()` and `waitpid()` calls determines what child (or children) the parent process waits for.

The `options` argument in `wait3()` and `wait4()` contains a bitwise OR of any of the half a dozen options. You can see these options and their meaning by viewing the man

TABLE 19.4 Values of the `wpid` Parameter for the `wait4()` and `waitpid()` System Calls and Children Processes Waited for

wpid	The Caller Waits for
−1	Any child process
0	Any child process in the process group of the caller
> 0	The child with PID `wpid`
< −1	Any child process whose process group ID equals the absolute value of `wpid`

TABLE 19.5 The Equivalences of the `wait4()` System Calls

`wait4(..., ..., ..., 0);`	`waitpid(..., ..., ...);`
`wait4(-1, ..., ..., ...);`	`wait3(..., ..., ...);`
`wait4(-1, ..., 0, 0);`	`wait(...);`

pages for these calls and by browsing through the header file **/usr/include/sys/wait.h**. The man pages also describe several macros that you may use to determine the status of a terminated child process. For example, you can find out if a child process terminated due to the arrival of a signal and, if so, the number of the signal that caused the process to terminate. These macros are defined in **/usr/include/sys/wait.h**.

When the `WNOHANG` option is specified and no processes are there to report the status, `wait4()` and `wait3()` do not block, and return 0 as the PID. Table 19.5 shows a few equivalences of the `wait4()` call and other calls in the `wait()` class.

The **fork.c** program in the following session uses the `fork()` system call to create a child process. The child process terminates after displaying its PID and its parent PID. The parent process waits for the child process and terminates after displaying its PID and the child's PID. Note that the PID displayed by the parent is, as expected, the same as the parent PID displayed by the child process.

```
% cat fork.c
#include <unistd.h>

extern int errno;

int main(void)
{
       int pid, status;

       pid = fork();
       if (pid == -1) {
           perror("Fork failed.");
           exit(1);
       }
       /* Child process */
       if (pid == 0) {
           printf("\nCHILD: Child here with PID = %d.\n",
                  getpid());
           printf("CHILD: My Mom has PID = %d.\n", getppid());
```

```
                exit(0);
        }
        /* Parent process */
        wait(&status);
        printf("\nPARENT: Mom here with PID = %d.\n", getpid());
        printf("PARENT: Well done, child!\n\n");
        exit(0);
}
% gcc48 -w fork.c -o fork
% ./fork
CHILD: Child here with PID = 28375.
CHILD: My Mom has PID = 28374.

PARENT: Mom here with PID = 28374.
PARENT: Well done, child!
%
```

19.3.5 Overwriting a Process Image

A process may overwrite itself with another executable image. The `execve()` and `fexecve()` system calls may be used to do so. These calls differ from each other in the manner in which parameters are passed to the caller process. Here are brief descriptions of these calls.

```
#include <unistd.h>
int execve(const char *path, char *const argv[], char *const envp[]);
int fexecve(int fd, char *const argv[], char *const envp[]);
```

Success: Control does not return to the caller process because it has been overwritten with a new executable
Failure: –1 and kernel variable `errno` set to indicate the type of error

In the `execve()` call, `path` is the pathname of the ordinary file that contains the executable image of the new process. In the case of the `fexecve()` system call, `fd` is the file descriptor of the file that contains the image of the new process. The file may contain the binary executable code or a script to be executed by an interpreter, such as a shell or Perl script. A script file begins with a line in the following format:

```
#! interpreter [arg]
```

The `execve()` system call actually overlays the caller process with `interpreter` and passes the script file to it as an argument. If the `arg` parameter is not specified on the first line, the script file is specified as the first argument to `interpreter`; otherwise, it is specified as the second argument. The `argv` argument is a pointer to a null-terminated array of pointers to null-terminated strings that become the command line arguments of the script or the binary executable. The `envp` argument is structurally similar to `argv` and contains the values of different environment variables.

If the file that is to overlay the caller process is a binary executable, it is executed just like a main program in C with command line arguments is executed, as in:

```
int main(int argc, char **argv, char **envp)
```

where `argc` is the number of command line arguments including the program name, and `argv` and `envp` are as described in the previous paragraph.

File descriptors open in the caller process remain open in the new process, except for those descriptors for which the `close-on-exec` flag is set. Signals set to be ignored in the caller process remain ignored in the new process and signals set to be caught in the caller process are set to default action in the new process. The new process also inherits the following identities and attributes of the caller process: PID, PPID, PGID, working directory, root directory, control terminal, access groups, resource usages, timers, resource limits, signal mask, and umask.

The `execve()` and `fexecve()` system calls may fail for different reasons, some of which are listed in Table 19.6.

The system call interface for overwriting a process with an executable image specified by the `execve()` and `fexecve()` calls is quite cumbersome. UNIX provides several library functions with easier interfaces for performing the same task. The simplest of these is the `execl()` library call, which is the front end of the `execve()` system call. We use this call in the examples that we discuss in this book. Here is a brief description of the call.

```
#include <unistd.h>
int execl(const char *path, const char *arg, ... /*, (char *)0 */);
```

Success: Control does not return to the caller process because it has been overwritten with a new executable
Failure: –1 and kernel variable `errno` set to indicate the type of error

The `execl()` call may fail for the same reasons that the `execve()` system call fails.

The **fork_exec_1.c** program shown in the following session creates a child process that overwrites itself with the executable in the **/bin/date** file to display today's date and the

TABLE 19.6 Some of the Reasons for the `execve()` or `fexecve()` System Call to Fail

Reason for Failure	Value of `errno`
A component in the `path` argument, except the last component, is not a directory.	ENOTDIR
The ordinary file specified as the last component in the `path` argument is not found.	ENOENT
Search permission for a directory in `path` has not been given. The last component in `path` is not an ordinary file. The last component in `path` does not have the execute permission on.	EACCES
The ordinary file does not have a valid magic number in its header.	ENOEXEC
The process requires more virtual memory than the system-imposed limit.	ENOMEM
One or more of the three arguments of the call point to an illegal address.	EFAULT
An error occurred while reading the file.	EIO
The `fd` argument is not a valid file descriptor.	EBADF

current time. Thus, the child becomes the date process. The parent process displays the PID of the child process, waits for the child process to terminate, and displays the PID of the terminated child process. The sample run of the program shows that, as expected, the PID displayed by the parent for the child process and that of the terminated child process are the same.

```
% cat fork_exec_1.c
#include <unistd.h>

extern int errno;

int main(void)
{
       int pid, status;

       pid = fork();
       if (pid == -1) {
           perror("Fork failed");
           exit(1);
       }
       /* Child process */
       if (pid == 0) {
           execl("/bin/date", "date", (char *) NULL);
           perror("execl failed");
           exit(1);
       }
       /* Parent process */
       printf("Child PID = %d.\n", (int) pid);
       pid = wait(&status);
       printf("Child with PID %d has terminated.\n", (int) pid);
       exit(0);
}
% gcc48 -w fork_exec_1.c -o forke1
% ./forke1
Child PID = 37831.
Sun Mar 15 10:04:49 PKT 2015
Child with PID 37831 has terminated.
%
```

The **fork_exec_2.c** program creates a child process that creates a directory, called **dir1**, in the current directory using the execl() call. The parent process creates a file called **foo** in **dir1**, opens this file, writes the Hello, world! message into **foo**, resets the file pointer to the beginning of the file, reads back the Hello, world! message from **foo**, displays this message on standard output, removes **foo** from **dir1**, and removes the now empty **dir1**. The compilation and a sample run of the program are shown in the following session.

```
% cat fork_exec_2.c
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>

#define SIZE 512
#define MODE 0644
#define Message "Hello, world!\n"

extern int errno;

int main(void)
{
        int n, nr, nw, fd, pid, status;
        char buf[SIZE];

        pid = fork();
        if (pid == -1) {
            perror("Fork failed");
            exit(1);
        }
        /* Child process */
        if (pid == 0) {
            execl("/bin/mkdir", "mkdir", "dir1", (char *) NULL);
            perror("execve failed.\n");
            exit(1);
        }
        /* Parent process */
        while (((pid = wait(&status)) == -1) && errno == EINTR)
            ;
        if (pid == -1) {
            perror("Wait failed");
            exit(1);
        }
        /* Open or create dir1/foo file */
        if ((fd = open("dir1/foo", O_RDWR|O_CREAT, MODE)) == -1) {
            perror("Open failed");
            exit(1);
        }
        /* Write to foo */
        if ((nw = write(fd, Message, strlen(Message))) == -1 ) {
            perror("Write failed");
            exit(1);
        }
        /* Rewind file pointer */
        n = lseek(fd, 0L, SEEK_SET);
        /* Read back from foo */
        if ((nr = read(fd, buf, nw)) == -1) {
            perror("Read failed");
            exit(1);
        }
        /* Throw on standard output */
        write(1, buf, nr);
        if (close(fd) == -1) {
            perror("File closing");
            exit(1);
        }
        /* Remove dir1/foo to make dir1 an empty directory */
        unlink("dir1/foo");
        /* Remove the now empty directory dir1 */
        rmdir("dir1");
        exit(0);
}
% gcc48 -w fork_exec_2.c -o forke2
% ./forke2
Hello, world!
%
```

EXERCISE 19.10

Compile and run the preceding program on your system. Does it produce the expected output?

19.3.6 Creating a Zombie Process

A process whose parent is not waiting (i.e., is sleeping or has finished execution) when it terminates cannot report its termination status to its parent and is called a *zombie process*. It has completed its work but some of the resources allocated to it may not be returned to the system. Eventually, init, the grandparent of all user processes, takes over the parenthood of the zombie process, receives its status, and releases the remaining system resources.

The following piece of code spawns a child process and puts the parent process to sleep for 10 seconds. The child process displays its PID and exits. Because the parent process is not waiting when the child terminates, the child process becomes a zombie. In the sample run, we suspend the parent process by pressing `<Ctrl+Z>` and use the `ps` command to display the status as zombie. Note that the child process is marked as `<defunct>` and its status is `Z` (for zombie). We bring the zombie process into the foreground by using the `fg` program.

```
% cat create_zombie.c
int main(void)
```

```
{
       int pid;

       pid = fork();
       if (pid == -1) {
           perror("Fork failed");
           exit(1);
       }
       /* Child process */
       if (pid == 0) {
           printf("Child's PID  = %d\n", getpid());
           exit(0);
       }
       /* Parent process */
       printf("Parent's PID = %d\n", getpid());
       sleep(10);
       /* Parent process does not wait for the child */
       /* process and child becomes a zombie process */
       exit(0);
}
% gcc48 -w create_zombie.c -o create_zombie
% ./create_zombie
Parent's PID = 14978
Child's PID  = 14979
<Ctrl+Z>
Suspended
% ps
  PID TT  STAT    TIME COMMAND
14522  1  Ss   0:00.42 -csh (csh)
14978  1  T    0:00.00 ./create_zombie
14979  1  Z    0:00.00 <defunct>
14986  1  R+   0:00.01 ps
% fg
zombie
%
```

A sample run under Solaris is shown in the following session. Note that you need to use the ps –al command to display the status of the zombie processes.

```
% ./create_zombie
Parent's PID = 12295
Child's PID  = 12296
<Ctrl+Z>
Stopped (user)
```

```
% ps -al
 F S   UID   PID  PPID C PRI  NI  ADDR    SZ WCHAN TTY    TIME  CMD
 0 Z  1007 12296 12295 0   0   -     -     0     - ?      0:00  <defunct>
 0 T  1007 12295 12294 0  40  20     ?   432       pts/1 0:00 ./create_zombie
 0 O  1007 12297 12294 0  40  20     ? 2318       pts/1 0:00  ps
 0 S  1007 12294 12166 0  40  20     ? 2179     ? pts/1 0:00  csh
% fg
zombie
%
```

When a child of the init process terminates, init calls one of the `wait()` system calls to collect the exit status of the child. Thus, none of the children of init ever becomes a zombie. The child may be a process that was directly created by init or was inherited by init because the parent of the process had died (or was not waiting) when the process terminated.

EXERCISE 19.11

Replicate the preceding session on your system. Does the **create_zombie.c** program work as expected?

19.3.7 Terminating a Process

A process can use the `kill()` system call to terminate a process for which it has such permission. A process owned by the superuser may terminate any process. We discuss this system call in detail in Section 19.5.4.

19.4 PROCESSES AND THE FILE DESCRIPTOR TABLE

We now discuss the importance of PPFDT in relation to file sharing between processes, I/O redirection, and communication between processes using different channels including pipes, FIFOs, and sockets. We discuss the concept of *file sharing* in Section 19.4.1 and *I/O redirection* in Section 19.4.2. Chapter 20 describes UNIX interprocess communication in detail.

19.4.1 File Sharing between Processes

As discussed earlier, a child process gets a copy of the PPFDT of its parent. It means that if a process opens a file before creating a child process, the parent and child processes have access to the file descriptor of the open file. Consequently, the I/O performed on the open file by either process is visible to the other process.

The program **file_sharing.c** illustrates how parent and child processes share files that are open before the child process is created. We demonstrate the concept with only one file, but it extends to multiple files. The program takes `filename` and `nbytes` (the I/O size in bytes) as command line arguments and opens for reading and writing the file specified as the command line argument. It then creates a child process that inherits the PPFDT and, therefore, the file descriptor of the opened file. Both the parent and child processes do I/O with the same file. The program displays the data read by both processes and the position of the file pointer after I/O in both processes, verifying that both processes do I/O with the same file. The `nbytes` argument may vary from 0 to 512 because the size of the read/write buffer has been set to 512 bytes. We use the `sleep(1)` call in the parent process to make sure that the child process performs I/O before the parent process. It is done to make sure that the values of the file pointer are displayed correctly in both processes. A few sample runs follow the listing of the program to verify that the file-sharing concept works as discussed. Note that **foo** is the name of an existing file.

```
% cat file_sharing.c
/* file_sharing.c: fileshare filename nbytes */

#include <fcntl.h>
#include <unistd.h>
#include <sys/wait.h>

#define SIZE 512

int main(int argc, char *argv[])
{
       int fd, n, nr, nbytes, pid, status;
       char buf[SIZE];

       /* Open file */
       if (argc != 3) {
           printf("%s filename nbytes\n", argv[1]);
           exit(1);
       }
       nbytes = atoi(argv[2]);
       if ((fd = open(argv[1], O_RDWR)) == -1) {
           perror("File opening");
           exit(1);
       }
       /* Create a child */
       pid = fork();
       /* Child process */
       if (pid == 0) {
           /* Read and display on stdout */
           if ((nr = read(fd, buf, nbytes)) == -1) {
               perror("Read failed");
               exit(1);
           }
           write(1, buf, nr); /* Throw on standard output */
           n = lseek(fd, 0L, SEEK_CUR);
           printf("\n\nCHILD: position of the file pointer after
                  I/O is %d\n\n", n);
           close(fd);
```

```
            exit(0);
        }
        /* Parent process */
        sleep(1); /* Allow the child process to do I/O first. */
        /* Read and display on stdout */
        if ((nr = read(fd, buf, nbytes)) == -1) {
            perror("Read failed");
            exit(1);
        }
        write(1, buf, nr); /* Throw on standard output */
        n = lseek(fd, 0L, SEEK_CUR);
        printf("\n\nPARENT: position of the file pointer after I/O
               is %d\n\n", n);
        close(fd);
        while (wait3(&status, WNOHANG, 0) >= 0)
               ;
        exit(0);
}
% gcc48 -w file_share.c -o fileshare
% ./fileshare foo 32
The sh utility is the standard c

CHILD: position of the file pointer after I/O is 32

ommand interpreter for the syste

PARENT: position of the file pointer after I/O is 64

% ./fileshare foo 64

The sh utility is the standard command interpreter for the syste

CHILD: position of the file pointer after I/O is 64

m. The current version of sh is close to the IEEE Std 1003.1 (?

PARENT: position of the file pointer after I/O is 128

% ./fileshare foo 256
The sh utility is the standard command interpreter for the system.
The current version of sh is close to the IEEE Std 1003.1
("POSIX.1") specification for the shell. It only supports features
designated by POSIX, plus a few Berkeley extensions.

CHILD: position of the file pointer after I/O is 256

This man page is not intended to be a tutorial nor a complete
specification of the shell.
The shell is a command that reads lines from either a file or the
terminal, interprets them, and generally executes other commands.
It is the program that is sta

PARENT: position of the file pointer after I/O is 512
%
```

EXERCISE 19.12

Replicate the preceding session on your system. Does the **file_sharing.c** program work as expected?

EXERCISE 19.13

Modify the **file_sharing.c** program so that it takes the number of bytes to be read as the first command line argument and the filename as the second argument. Compile and run the program with different files and bytes-to-read as command line arguments.

19.4.2 Duplicating File Descriptor

You can use the `dup()` and `dup2()` system calls to duplicate a file descriptor. A UNIX shell uses these system calls to implement I/O redirection. Here are brief descriptions of these calls.

```
#include <unistd.h>
int dup(int olddes);
int dup2(int olddes, int newdes);
```

Success: A newly allocated file descriptor
Failure: –1 and kernel variable `errno` set to indicate the type of error

The `dup()` call duplicates the existing file descriptor `olddes` in the PPFDT and returns the duplicated file descriptor. As discussed earlier, the new file descriptor is the smallest unused descriptor in the PPFDT. The old and new file descriptors point to the same entry in the SFT. Thus, changes in the read/write file pointer, file contents, and file attributes are visible through both descriptors. This is true regardless of the file type that the descriptor points to: ordinary file, pipe, socket, or FIFO.

The `dup2()` call is similar to `dup()`, except that the newly allocated descriptor is explicitly specified as the second argument. If `olddes` is a valid descriptor and is equal to `newdes`, the call is successful and performs a null operation; that is, it does not do anything. If `oldfd` is a valid descriptor and is not equal to `newdes`, the `dup2()` call first deallocates `newdes` and then performs the duplicate operation. If `olddes` is not a valid descriptor, the call fails and no duplication operation is performed.

These calls may fail for the reasons specified in Table 19.7.

These calls are normally used to implement input, output, and error redirection for a process (see Chapter 9). You can do input redirection by closing standard input for the

TABLE 19.7 Reasons for the `dup()` and `dup2()` System Calls to Fail

Reason for Failure	Value of `errno`
The descriptor specified in `olddes` is not a valid active descriptor (i.e., has not been allocated using a system call such as `open()`) or the descriptor specified in `newdes` is greater than the size of PPFDT or is a negative number	`EBADF`
The process has already used the maximum number of descriptors (i.e., the PPFDT is being used to full capacity)	`EMFILE`

process, opening the file to which input has to be redirected, and attaching standard input to the file by duplicating the file descriptor to the PPFDT slot for standard input. Similarly, you can do output and error redirection for a process.

The following program shows sample uses of the `dup()`system call. The program takes a file as a command line argument, opens the file for writing, and saves the file descriptor for the file in `fd`. If the file does not exist, the program creates it and sets its access permissions to read and write. The program then closes the file descriptor for standard output by using the `close(1)` system call. After the standard file descriptor has been closed, slot number 1 of the PPFDT becomes free and the `dup(fd)` system call copies the entry for `fd` in the PPFDT into slot 1. After this has been done, anything directed to standard output is sent to the opened file. Thus, the output of the `printf()` call is redirected to the file passed as the command line argument. In the following session, we pass **foo** as the command line argument to the program. The output of the `cat  foo` command verifies that the standard output has been redirected to **foo**.

```
% cat dup.c
#include <fcntl.h>
#include <sys/stat.h>

int main(int argc, char *argv[])
{
    int fd;

    /* Open file */
    if (argc == 1) {
              printf("No file specified as command line
                     argument.\n");
              exit(1);
    }
    if ((fd = open(argv[1], O_WRONLY|O_CREAT |O_TRUNC, S_IREAD|S_
                   IWRITE)) == -1) {
              perror("File opening");
              exit(1);
    }
    /* Close standard output */
    close(1);
    /* Duplicate fd into file descriptor 1, i.e., stdout */
    if (dup(fd) == -1) {
              perror("Duplicating file descriptor");
              exit(1);
    }
    /* Close fd in order to release the extra slot in the PPFDT */
    if (close(fd) == -1) {
              perror("File closing");
              exit(1);
    }
```

```
        /* Stdout redirected to the file passed as command line
           argument */
        printf("Hello, world!\n");
        exit(0);
}
% gcc48 -w dup.c -o dup
% ./dup foo
% cat foo
Hello, world!
%
```

In the following session, we show the use of `dup2()`in the **dup2.c** program to perform the same task as performed by the **dup.c** program—that is, redirect standard output to the file passed to the program as a command line argument.

```
% cat dup2.c
#include <fcntl.h>
#include <sys/stat.h>

int main(int argc, char *argv[])
{
        int fd;

        /* Open file */
        if (argc == 1) {
                printf("No file specified as command line
                        argument.\n");
                exit(1);
        }
        if ((fd = open(argv[1], O_WRONLY|O_CREAT|O_TRUNC,
                       S_IREAD|S_IWRITE)) == -1) {
                perror("File opening");
                exit(1);
        }
        /* Duplicate fd into file descriptor 1, i.e., stdout */
        if (dup2(fd, 1) == -1) {
                perror("Duplicating file descriptor");
                exit(1);
        }
        /* Close fd in order to release the extra slot in the PPFDT */
        if (close(fd) == -1) {
                perror("File closing");
                exit(1);
        }
        /* Stdout redirected to the file passed as command line
        argument */
        printf("Hello, world!\n");
        exit(0);
}
% gcc48 -w dup2.c -o dup2
% ./dup2 foo
% cat foo
Hello, world!
%
```

You can redirect standard input (or standard error) in the same manner as discussed. All three standard files may be redirected to one file or multiple files, one each for standard input, standard output, and standard error.

EXERCISE 19.14

Replicate the preceding sessions on your system. Do the **dup.c** and **dup2.c** programs work as expected?

19.5 GETTING THE ATTENTION OF A PROCESS: UNIX SIGNALS

Similar to how a peripheral device may use an interrupt to get the attention of the CPU and be served, you may use a signal to get the attention of a process. We discussed this topic in detail in Chapters 13 and 15 to explain signal/interrupt handling in shell scripts. Here, we discuss signal handling in a C program.

19.5.1 What Is a Signal?

As stated earlier, a signal in UNIX vocabulary is an event that interrupts the execution of a process. We discussed the events that cause signals in Chapter 13, Section 13.4 while discussing advanced Bourne Shell programming. Some of the events that cause signals are listed in Table 13.1. We reproduce that table here as Table 19.8.

You can view the complete list of events that cause signals by viewing the man pages for the `kill` command or `kill()` system call. You can also use the `kill –l` command to display the list of signals. A similar list is also found in the **/usr/include/sys/signal.h** file.

You can send a signal to a process by using the `kill` command or the `kill()` system call. Both take two arguments, a signal number and the PID of the process to receive the signal. For example, the `kill -16 12345` command sends signal number 16 to the process with PID 12345.

19.5.2 Intercepting Signals

A process may use the `sigaction()` system call or the `signal()` library call to intercept a signal and take one of the following three possible actions:

1. Ignore the signal
2. Take the default action as defined by the kernel
3. Take the programmer-defined action

TABLE 19.8 Commonly Used Signals, Their Number, and Their Purpose

Signal	Number	Purpose
`SIGHUP` (hang up)	1	Informs the process when the user who ran the process logs out, and the process terminates
`SIGINT` (keyboard interrupt)	2	Informs the process when the user presses `<Ctrl+C>` and the process terminates
`SIGQUIT` (quit signal)	3	Informs the process when the user presses `<Ctrl+|>` or `<Ctrl+\>` and the process terminates
`SIGKILL` (sure kill)	9	Terminates the process with no further processing (e.g., exception handling) when the user sends this signal to it with the `kill -9` command
`SIGSEGV` (segmentation violation)	11	Terminates the process upon memory fault when a process tries to access memory space that does not belong to it
`SIGTERM` (software termination)	15	Terminates the process when the `kill` command is used without any signal number
`SIGTSTP` (suspend/stop signal)	18	Suspends the process; usually `<Ctrl+Z>`
`SIGCHLD` (child finishes execution)	20	Informs the process of termination of one of its children

TABLE 19.9 Reasons for the Library Call `signal()` to Fail

Reason for Failure	Value of `errno`
The signal specified in `sig` is not a valid signal number	`EINVAL`
The process tried to ignore or specified a handler for the `SIGKILL` or `SIGSTOP` signal	`EINVAL`

The interface for the `sigaction()` system call is rather complex. The library call `signal()` has a much simpler interface. For this reason, programmers normally use `signal()` for handling signals. Here is a brief description of this call.

```
#include <signal.h>
sign_t signal(int sig, sig_t func);
```

Success: `0`
Failure: −1 and kernel variable `errno` set to indicate the type of error

The `func` argument is `SIG_IGN` for ignoring a signal and `SIG_DFL` for the default, kernel-defined action. A call to the `signal()` function may fail for two reasons, as shown in Table 19.9.

19.5.3 Setting Up an Alarm

The library call `alarm()` sends the `SIGALRM` signal to the calling process after the specified number of seconds. Thus, the call `alarm(10)` sends the `SIGALRM` signal to the caller process after 10 seconds. Here is a brief description of the call:

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

Success: 0 if no alarm is currently set and the number of seconds left on the timer of the previous `alarm()`
Failure: –1 and kernel variable `errno` set to indicate the type of error

The maximum number of seconds allowed is 100,000,000. If an alarm has already been set but has not gone off (i.e., the signal has not been sent to the process), another call to alarm supersedes the previous call. The `alarm(0)` call cancels the current alarm and `SIGALRM` is never delivered to the calling process.

The **signals.c** program shown in the following session ignores the `SIGHUP` signal, and takes programmer-defined actions for `SIGINT` (signal number 2, `<Ctrl+C>`) and `SIGALARM` (signal number 14). For all other signals, the process takes the system-defined default action.

```
% cat signals.c
#include <sys/signal.h>

#define TRUE 1

void nicetry(void);
void onalarm(int);

int main(void)
{
        signal(SIGHUP, SIG_IGN);
        signal(SIGINT, nicetry);
        signal(SIGALRM, onalarm);
        alarm(10);
        while (TRUE) {
           printf("Waiting for alarm.\n");
           sleep(9);
        }
}

void nicetry(void)
{
   printf("Nice try! Sorry you cannot terminate me like this.\n");
}

void onalarm(int signal)
{
   printf("Caught signal number %d. Going back to work.\n", signal);
}
%
```

In the following session, we show the compilation and working of the code by running the program. Note that when we press `<Ctrl+C>` (which appears as `<Ctrl+C>` in the shell session), the program responds with the message displayed by the `nicetry()` function. In order to test that the response to `SIGHUP` also works as expected, we put the `signals` process (PID 19156) in the background using `<Ctrl+Z>` (which appears as `<Ctrl+Z>` in the shell session) and send the `SIGHUP` signal to the process using the `kill -1 19156` command. The handling of `SIGALRM` also works as expected when the program displays the message `Caught signal number 14. Going back to work.`

```
% gcc48 -w signals.c -o signals
% ./signals
Waiting for alarm.
<Ctrl+C>Nice try! Sorry you cannot terminate me like this.
Waiting for alarm.
<Ctrl+C>Nice try! Sorry you cannot terminate me like this.
Waiting for alarm.
Caught signal number 14. Going back to work.
Waiting for alarm.
<Ctrl+Z>
Suspended
% ps
  PID TT  STAT    TIME COMMAND
17975  1  Ss   0:00.40 -csh (csh)
19156  1  T    0:00.00 ./signals
19199  1  R+   0:00.01 ps
% kill -1 19156
Waiting for alarm.
% ps
  PID TT  STAT    TIME COMMAND
17975  1  Ss   0:00.41 -csh (csh)
19156  1  S    0:00.00 ./signals
19234  1  R+   0:00.01 ps
%
Waiting for alarm.
Waiting for alarm.
<Enter>
% kill -9 19156
<Enter>
[1]    Killed                       signals
% ps
  PID TT  STAT    TIME COMMAND
17975  1  Ss   0:00.44 -csh (csh)
19286  1  R+   0:00.01 ps
%
```

On some UNIX systems, including Solaris, the settings for intercepting signals using the `signal()` call is effective only once. When a signal has been intercepted and handled according to signal settings, the signal settings have to be reestablished in order for signal handling to work correctly. For this purpose, the relevant signal handlers include the code for resetting signals for future signal handling. For the **signals.c** program to work correctly on such systems, you need to modify the `nicetry()` function, the handler for SIGINT, as follows:

```
void nicetry(void)
{
    signal(SIGINT, nicetry);
    printf("Nice try! Sorry you cannot terminate me like this.\n");
}
```

EXERCISE 19.15

Replicate the preceding sessions on your system. Does the **signals.c** program work as expected?

Several Internet services such as ftp are offered via server processes that provide services to client processes through multiple *slave processes*, one for each client. Slaves are precreated and/or created dynamically by the main server when needed. The main server is also known as the *master server*. Such servers are known as *multiprocess servers* or *concurrent servers*. A concurrent server runs in an infinite loop with the following code structure:

```
while (1) {
   wait for a client request
   create a slave process when a client request arrives
   slave handles the client request
}
```

Because concurrent servers keep creating slave processes as client requests arrive, it is important to terminate a slave process properly and remove it from the system after it has provided its service to a client process. This is done by using the `exit()` and `wait()`calls in tandem, in the slave and master processes, respectively. If the master server process does not remove the slave processes after they have provided their services, a large number of zombie processes will be created in the system. We would want the server to spawn a child, wait, and then when returned to, kill the child, but we can't do that with a concurrent server since it has to continue to process further client requests.

Recall that when a process terminates, the UNIX kernel sends the `SIGCHLD` signal to its parent. A concurrent server process uses this feature to intercept all `SIGCHLD` signals to remove the terminating slave processes from the system by using the following code structure:

```
...
int main(...)
{
   ...
```

```
    signal (SIGCHLD, zombie_gatherer);
    while(1) {
       wait for a client request
       create a slave process when a client request arrives
       slave handles the client request
    }
    ...
}
void zombie_gatherer(int signal)
{
int status;
while (wait3(&status, WNOHANG, 0) >= 0)
    ;
}
```

Recall that the `WNOHANG` option for the `wait3()` system call makes it a nonblocking call, in the sense that if it does not find a child process that has performed `exit()`, it returns –1. When `wait3()` returns –1, the control returns from the `zombie _ gatherer()` function to the line of code in the main function that was interrupted by `SIGCHLD`.

We discuss the algorithms for the various types of Internet servers in Chapter 20.

19.5.4 Sending Signals

A process can send a signal to another process by using the `kill()` system call. Here is a brief description of the call.

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

Success: 0
Failure: –1 and kernel variable `errno` set to indicate the type of error

If `pid` is 0, the signal is sent to all the processes whose group ID (GID) is equal to the sender process and for which the sender process has the permission to do so. If the sender process has superuser privileges and `pid` is –1, the specified signal is sent to all the processes having the same UID as the sender, the system processes, and the init process (PID = 1).

The `kill()` call may fail for the reasons listed in Table 19.10.

TABLE 19.10 Reasons for the `kill ()` System Call to Fail

Reason for Failure	**Value of `errno`**
The signal specified in `sig` is not valid	EINVAL
The process specified in `pid` does not exist	ESRCH
The process using `kill ()` does not have the permission to send the signal to the process specified in `pid`	EPERM

The **killer.c** program in the following session takes a PID and a signal number as command line arguments and uses the `kill()` system call to send the specified signal to the process with the given PID. After compiling the program and saving the executable code in the file called `killer`, we start a C shell process, and use the `ps` command to see the PID of the new C shell process. We then use the `killer` program to send signal number 2 (`SIGINT`) to the new C shell process (PID 53426). The output of the `ps` command shows that, as expected, the C shell does not terminate. When we send signal number 9 (`SIGKILL`; sure kill) to the new C shell process, the shell terminates. The `SIGHUP` signal would also terminate the shell process.

```
% cat killer.c
/* killer.c: killer signal pid */

#include <sys/types.h>

int main(int argc, char *argv[])
{
       pid_t pid;
       int signal;

       if (argc != 3) {
           printf("Inappropriate number of command line
                  arguments.\n");
           exit(0);
       }
       pid = atoi(argv[1]);
       signal = atoi(argv[2]);
       if (kill(pid, signal) == -1) {
           perror("Kill failed");
           exit(1);
       }
       exit(0);
}
% gcc48 -w killer.c -o killer
% csh
% ps
  PID TT  STAT    TIME COMMAND
51926  1  Ss   0:00.29 -csh (csh)
53426  1  S    0:00.28 csh
53435  1  R+   0:00.00 ps
% ./killer 53426 2
% ps
  PID TT  STAT    TIME COMMAND
51926  1  Ss   0:00.29 -csh (csh)
53426  1  S    0:00.28 csh
53435  1  R+   0:00.00 ps
% ./killer 53426 9
```

```
Killed
% ps
  PID TT  STAT     TIME COMMAND
51926  1  Ss    0:00.29 -csh (csh)
53479  1  R+    0:00.01 ps
%
```

EXERCISE 19.16

Replicate the preceding sessions on your system. Make sure that the **killer.c** program works as expected.

19.6 IMPORTANT WEB RESOURCES

Table 19.11 lists some useful websites for UNIX system programming and related topics.

SUMMARY

We described the UNIX system calls and their use in small programs for process creation, process termination, and process management. Before discussing the system calls we explained in detail the concept of processes and threads. We then discussed how they differ from each other and the concept of user- and kernel-level threads. We also discussed the following concepts and kernel data structures in UNIX: the process control block, memory and disk images of processes, zombie processes, signals, and signal handling.

We discussed process management using the following system calls and library functions: `fork()`, `vfork()`, `exit()`, `_exit()`, `wait()`, and its different variants, the `getpid()`, `getppid()`, `signal()`, `dup()`, `dup2()`, `kill()`, `execve()`, `fexecve()`, and `execl()` calls. We discussed the concept of zombie processes and used the `exit()` and `wait()` calls to illustrate how a zombie process can be created. We also discussed the concept of file sharing between the parent and children processes.

Throughout the chapter, we showed the use of the various system calls and library functions in small C programs to illustrate various process management concepts.

TABLE 19.11 Web Resources for the UNIX System Programming and Related Topics

`https://www.freebsd.org/`	Home page for FreeBSD. Contains a lot of useful material, including FreeBSD source, manual pages, support, SVN repository, forums, user groups, etc
`https://computing.llnl.gov/tutorials/pthreads/`	An excellent tutorial on Pthreads by Blaise Barney, Lawrence Livermore National Laboratory

QUESTIONS AND PROBLEMS

1. What is a thread? What are user threads? What are kernel threads?
2. List the differences between processes and threads.
3. Suppose a system supports multithreaded processes with user-level threads. Where is the scheduling of threads carried out?
4. What do we mean by *race condition* in the context of multithreaded processes? What does a programmer need to do to handle the issue of race condition?
5. Explain the difference between single- and multithreaded kernels by using an example.
6. Why is a separate function needed to terminate a thread? Why can we not terminate a thread with the `_ exit()` system call or the `exit()` library function?
7. What is *atomic execution* of a piece of code?
8. What is a critical section? What is the critical section problem?
9. The Bakery Algorithm is an elegant solution for the N-process critical section problem. Browse the Web or read a book on operating system principles and concepts to find out the name of the author of this algorithm. Who is the author?
10. A process is said to be a program in execution. What really comprises a process in terms of the system resources that a process utilizes?
11. What is the process control block (PCB)? What are the names of the two parts of a process's PCB in UNIX? What kind of information do they contain about the process?
12. The PCB of a process is not accessible in user mode. Why?
13. Which part of the PCB of a UNIX process always remains in the main memory? What information about the process does it contain?
14. Why is bss not part of the disk image of an executable file but is part of the process that is created when the file is executed?
15. The UNIX command `size` may be used to display the size in bytes of the code, data, and bss segments of an executable file. Show an example to illustrate your answer.
16. What is the relationship of shared libraries with dynamic linking? What are the advantages of dynamic linking over static linking? What are the disadvantages of dynamic linking?
17. By default, the gcc compiler use dynamic linking. What is the compiler option for creating an executable program using static linking? Which linking generates smaller executable code? Why?

18. Compile a program using static and dynamic linking and then display the program size using the `ls –l` command. Show your shell session.

19. Display the sizes of the code, data, and bss sections of the executable program generated in Problem 7. Show the shell session. Why is the sum size of the code, data, and bss sections generated by the `size` command not equal to the size of the same executable generated by using the `ls –l` command?

20. Consider the following C programs and answer the questions that follow:

```
% cat bss_size_1.c
int BigData[1000000];
void main(void)
{
 BigData[0] = 0;
}
% cat bss_size_2.c
int BigData[1000000] = {1, 2, 3};
void main(void)
{
 BigData[0] = 0;
}
```

   a. Which of these two programs takes more disk space? Why?

   b. If executable codes are generated for the two programs, which would take more disk space? Which would require more main memory to execute? Explain your answers.

   c. What are the sizes of the bss and data segments of the executable files for these programs? What commands did you use to obtain these sizes? Show your shell session.

21. What is the purpose of the magic number of a file in UNIX? What command uses the magic number of a file to display its output? Show an example to illustrate your answer.

22. Write a small program that creates three children processes, `P1`, `P2`, and `P3`. The parent and each child display their PIDs and their parent PPIDs. The parent should display its PID and PPID only when all three of its children have terminated.

23. How many processes does the following C program create? Assume that all three `fork()` calls are successful. Draw the process tree for the program after the third `fork()` call has executed successfully. Explain your answer.

```
#include <stdio.h>

int main(void)
{
   int pid1, pid2, pid3;
```

```
    pid1 = fork();
    pid2 = fork();
    pid3 = fork();
}
```

24. Browse the man pages for the `vfork()` and `rfork()` system calls. What is the purpose of each of these calls? How does `vfork()` differ from `fork()`?

25. Write a program that has one file opened by the parent process and one by the child. The child writes `Long live UNIX.` to both files, reads and displays the contents of its own file, closes its file, and returns (i.e., exits). The parent reads and displays the contents of its file, closes the file, and terminates. Does the output of the program make sense? Explain your answer.

26. Write a program in which a parent opens two files (**file1** and **file2**), writes `Long live UNIX.` to **file1**, and spawns a child. The child copies the contents of **file1** to **file2**, closes the two files, and returns (i.e., exits). The parent reads and displays the contents of the two files, closes them, and terminates. Does the output of the program make sense? Explain your answer.

27. When a process uses the `execve()` or `fexecve()` call to overwrite itself, signals set to be caught in the caller process are set to default action in the new process. Why are signals set to default action—why not the actions as specified in the caller process?

28. What are the differences between the `_ exit()` and `exit()` calls?

29. What would happen if you terminated a multithreaded process with the `exit()` call? Explain your answer.

30. What are the `wait4()` equivalents of the following calls? The three dots (...) represent the value of the corresponding argument for the calls:

    - `waitpid(..., ..., ...);`
    - `wait3(..., ..., ...);`
    - `wait(...);`

31. Write a program that creates a child process that displays its PPID and sleeps for 60 seconds. The parent displays its PID, the termination status of child, and the reason for the child's termination (i.e., the signal that caused its termination). Run the program in the background, use the `ps` command to identify the PID of the child process, and then terminate the process by sending it a signal using the `kill` command. Show your code and a few sample runs. Make sure that the program displays the correct values for the return code of the child as well as the signal number that caused its termination.

32. What is the purpose of the `kill()` system call? What happens when 0 or –1 is specified as the PID to this call?

33. What is the effect of the `kill(getpid(), SIGKILL)` system call?

34. Suppose you run a program under a C shell. What will be the effect of the `kill(getppid(), SIGKILL)` system call in the program?

35. What is the effect of executing the `dup()` system call? What happens if the specified descriptor is invalid?

36. What happens if `dup2()` is used and the old and new descriptors specified in the call are the same, provided that the old descriptor is valid? What happens if the old descriptor is not valid?

37. The **dup2.c** program discussed in Section 19.4.2 illustrates how a shell implements output redirection. Modify this program so that it takes two files as arguments and implements both input redirection and output redirection.

38. What is the difference between `fork()` and `vfork()`? As a system programmer, when should you prefer to use `vfork()` over `fork()`?

39. What is the output of the following program? Explain your answer.

```
int main(void)
{
  pid_t pid;
  int i=100, status;
  if ((pid = vfork()) == -1) {
     perror ("vfork failed");
     exit(1);
  }
  if (pid == 0) {
     i++;
     exit(0);
  }
  wait(&status);
  printf("%d\n",++i);
  exit(0);
}
```

40. What will be the output of the program listed in Problem 37 if the following statements are swapped? Explain your answer.

```
wait(&status);
printf("%d\n",++i);
```

Will the program always produce the same output every time it is run? Explain your answer.

41. Write a C program that creates a zombie process and then runs the `ps` command from within the process to verify that the process status is zombie. Show your program and its sample run.

42. Remove the `sleep(1);` statement in the **file_sharing.c** program discussed in Section 19.4.1. Compile the program and run it a few times. You will notice that for some sample runs, the file pointer value displayed by the parent and child processes is the same. Explain why this is so.
43. Identify the values of the following by browsing through the **<limits.h>** and **<sys/limits.h>** files on BSD and **<limits.h>** file on Solaris:
    - The maximum number of processes that can run on your system concurrently
    - The maximum number of children processes that a process can have at a given time
44. Use the `limit` command under the C shell or the `ulimit` command under Bash to determine the maximum number of processes that can run on your system concurrently and the maximum number of files that can be opened on your system concurrently.
45. Browse through the **<sys/signal.h>** file and identify all of the signals along with their numbers and their purpose.
46. What piece of code would you add to your code in the simple shell program that you wrote for Problem 30 in Chapter 18 so that the shell only terminates when the user presses `<Ctrl+D>`? It should not terminate due to a keyboard interrupt—that is, when you press `<Ctrl+C>`.
47. Give two reasons each for the following system calls to fail: `dup()`, `fork()`, `execve()`, `_ exit()`, `wait()`, `kill()`, and `signal()`.

CHAPTER 20

System Programming III

Interprocess Communication

Objectives

- To explain the concept of interprocess communication (IPC), important related system calls, macros, and data structures
- To describe IPC based on the client–server model
- To discuss the most primitive form of IPC between a parent process and its children processes
- To discuss IPC between related processes on a machine using UNIX pipes
- To describe IPC between unrelated processes on the same machine using UNIX named pipes (FIFOs)
- To discuss IPC between related or unrelated processes on the same machine or different machines on a network using sockets
- To elucidate synchronous and asynchronous IPC
- To discuss in detail the client–server models for socket-based IPC
- To describe in detail the design of client–server software and possible types of servers
- To study multiple-process-based concurrent servers
- To explain concurrent servers based on the `select()` system call
- To explain the UNIX superserver, `inetd`
- To briefly discuss concurrent clients
- To cover the system calls, library calls, commands, macros, and primitives

```
_exit(), FD_CLR(), FD_ISSET(), FD_SET(), FD_ZERO(), accept(),
bind(), bzero(), close(), connect(), exit(), gethostbyaddr(),
gethostbyname(), getservent(), gettablesize(), htonl(),
htons(), inet_addr(), inet_aton(), inet_ntoa(), inet_ntop(),
inet_pton(), listen(), memcpy(), memset(), mkfifo(), mknod(),
ntohl(), ntohs(), open(), perror(), pipe(), read(), recvfrom(),
sendto(), select(), shutdown(), signal(), sizeof(), socket(),
strtol(), system(), wait(), wait3(), waitpid(), write()
```

20.1 INTRODUCTION

UNIX interprocess communication (IPC) facilities exist so that processes and threads may communicate with one another and synchronize their actions. The three functional facilities into which IPC can be divided are communication, synchronization, and signals. Communication provides functionality to exchange data between processes or threads, synchronization provides for the synchronization of processes or threads, and signals are a communication technique where the signal number itself is a form of communication information. Although some of these facilities fundamentally exist for communication and synchronization, the term IPC is used to describe them all.

The IPC facilities provide similar functionality because of the following reasons:

1. Similar facilities evolved on different UNIX families, and then migrated between families. For example, FIFOs were developed on System V and (stream) sockets were developed on BSD.

2. New facilities have been developed to make up for deficiencies of similar earlier facilities. For example, the POSIX IPC facilities (message queues, semaphores, and shared memory) were designed to improve on the older System V IPC facilities. In particular, whereas POSIX IPC is thread safe, System V IPC is not.

Some facilities, which on the surface seem similar, are actually significantly different in terms of the methods and functionality they provide. For example, stream sockets can be used to communicate over a network, while FIFOs can be used only for communication between processes on the same machine. Of all of the IPC methods, only sockets permit processes to communicate over a network. Sockets are generally used in one of two domains: that which allows communication between processes on the same system, and on the Internet, which allows communication between processes on different hosts connected via a Transmission Control Protocol/Internet Protocol (TCP/IP) network. Often, only minor changes can convert a program that uses sockets between the two domains.

Our coverage of the topic is broadly divided into three types: (a) IPC between related processes on the same computer, (b) IPC between unrelated processes on the same computer, and (c) IPC between related or unrelated processes on the same or different computers on a network, including the Internet. The relationship between processes is normally a parent–child or sibling relation. We cover in detail the system calls, library functions, macros, data structures, and header files involved in creating the requisite communication channels, as well as using them for reading and writing messages between the processes involved in communication. After covering the preliminary topics, we focus on the discussion of UNIX IPC under the client–server paradigm and the fundamentals of Internet working with UNIX TCP/IP. In doing so, we describe the design of various types of clients and servers, including iterative and different types of concurrent servers based on slave processes and the `select()` system call. We also explain what a superserver is and how the UNIX superserver, `inetd`, works. Finally, we discuss concurrent clients. Throughout this chapter, we use the terms *computer*, *machine*, and *host* interchangeably.

20.2 IPC: COMMUNICATION CHANNELS AND COMMUNICATION TYPES

For communication between processes, you require a communication channel and communicating processes. The characteristics of communication channels are dependent on the location of processes, their relationship with each other, and the type of communication to be carried out between these processes. Processes may fall into the following types:

1. Related processes on the same computer
2. Unrelated processes on the same computer
3. Unrelated processes on different computers on a network

The communication channel may be categorized based on the direction of communication and the nature of messages through the channel. The communication channel may support only *simplex* (i.e., one-way) communication or it may support *full-duplex* (i.e., two-way/bidirectional) communication. The messages may be just a *stream of bytes* or messages with clear boundaries. Messages with clear boundaries may be of fixed or variable size but with a maximum size limit and, depending on the literature you read and the network layer at which they are handled, may be called *packets*, *frames*, *datagrams*, *segments*, *chunks*, and so on.

Depending on the algorithm/protocol used by them, the communicating processes may have to establish a connection with each other before starting communication, or they may just send messages by specifying the address/name of a process. The first style of communication is called *connection oriented* and results in guaranteed, in-order, error-free delivery of messages. The second style of communication is known as *connectionless* communication and results in best-effort delivery of messages but without any guarantees. The messages used in connection-oriented communication are usually without boundaries and those used in connectionless communication are usually with boundaries.

There are several communication channels that support IPC on UNIX machines, including the following:

1. Exit-wait
2. Pipe
3. Named pipe (FIFO)
4. Semaphore
5. Spin lock
6. Shared memory
7. Message queue
8. Sockets
9. System V transport layer interface (TLI)

We focus primarily on the following three commonly used channels: pipe, named pipe (FIFO), and socket. Which channel you should use in your applications depends on the relationship between communicating processes, their location, and the nature of application from the point of view of reliability of message delivery and the integrity of message content.

20.3 IPC: IMPORTANT SYSTEM AND LIBRARY CALLS, DATA STRUCTURES, MACROS, AND HEADER FILES

In this section, we describe briefly the most commonly used UNIX channels of communication between the three types of processes already mentioned (i.e., pipe, named pipe, and socket), related data structures, system calls, library functions, and macros. We will discuss them in detail later under the appropriate sections. Using the taxonomy of process types in Section 20.2, a *pipe* may be used for communication between type (a) processes, a *named pipe* (also called *FIFO*) for communication between type (a) processes or type (b) processes, and a *socket* for communication between processes belonging to any one of the three types. We describe in detail these channels as well as the system calls and library functions needed for input/output (I/O) through these channels in the remainder of this chapter. Table 20.1 shows a brief summary of important system calls required for IPC using pipes, named pipes (FIFOs), and sockets.

The list of commonly used macros and library calls in network-based IPC along with their purposes is shown in Table 20.2.

The list of important data structure in network-based IPC along with their purposes is shown in Table 20.3.

Details of these system calls, library functions, macros, and data structures are described at appropriate places in subsequent sections.

TABLE 20.1 Brief Summary of UNIX System Calls for IPC, Creating and Using Pipes, Named Pipes (FIFOs), and Sockets

System Call	Purpose
`pipe()`	Creates an IPC channel (two descriptors) for IPC between related processes on the same computer
`mkfifo()`	Creates an IPC channel (a file of a particular type) for IPC between related or unrelated processes on the same computer
`socket()`	Creates an endpoint (descriptor) for network-based IPC
`bind()`	Binds a local IP and protocol port number to a socket
`listen()`	Puts the socket in passive (listening) mode and sets the size of the queue where incoming connection requests may wait
`connect()`	Establishes a connection with a remote server
`accept()`	Accepts a client request for connection
`select()`	Waits for a connection request on a bit-set (flags) representing a set of descriptors for a specific period of time and flags for those descriptors that are ready for I/O
`sendto()`	Sends a datagram to a socket whose address has been prerecorded
`recvfrom()`	Receives a datagram and records the address of the sender socket
`read()`	Receives data (or datagram) from a socket on which `connect()` has been called
`write()`	Sends data (or datagram) to a socket on which `connect()` has been called
`close`	Terminates communication and deallocates a descriptor
`shutdown()`	Terminates TCP communication (I/O) in one or both directions

The list of important header files, specifically designed for IPC through pipes, FIFOs, and sockets, and used throughout the chapter, is briefly described in Table 20.4.

20.3.1 Byte Orders

Multibyte values may be stored, communicated, and manipulated in two orders: *little endian* and *big endian*. In the little endian order, the low-order byte of data value is stored in low storage location and the high-order byte in the high location. Thus, if the low-order byte is stored at memory location L, then the high-order byte is stored at location L + 1. The order is reversed in the big endian storage order; that is, the low-order byte of data value is stored in the high storage location (at address L + 1) and the high-order byte is stored in the low storage location (at address L). Figure 20.1 shows these storage orders in pictorial form.

Internet protocols deal with multibyte values in the big endian order. This means that network software stores and transmits multibyte data in the big endian order. Hence, the big endian order is also known as *network byte order*. Depending on the brand of central processing units (CPUs) used in the hosts on a network, they may use network byte order or little endian order. Intel processors use little endian order, whereas Sun and Motorola processors use big endian order. Thus, if a host uses an Intel CPU, it deals with multibyte values in little endian order.

The following program may be used to determine the byte order used by your computer system. Note that the compilation and sample run of the program shows that our UNIX system runs on a machine that uses little endian byte order. The `uname -p` command

TABLE 20.2 Commonly Used Macros, Library Calls, and Their Purpose

Macro	Purpose
`htons()`*	Converts a 16-bit value in host byte order to network byte order
`htonl()`*	Converts a 32-bit value in host byte order to network byte order
`ntohs()`*	Converts a 16-bit value in network byte order to host byte order
`ntohl()`*	Converts a 32-bit value in network byte order to host byte order
`bzero()`	Initializes a string to null (zero) bytes
`memset()`	Initializes a string to bytes of a particular character's value
`memcpy()`	Copies the given number of bytes from one string to another
`FD_SET()`**	Includes the given descriptor in the set (i.e., sets the relevant numbered bit to 1)
`FD_CLR()`**	Exclude the given descriptor in the set (i.e., sets the relevant numbered bit to 0)
`FD_ZERO()`**	Initializes a descriptor set to null set (i.e., all zeros)
`FD_ISSET()`**	Tests if a particular descriptor in the set is 0 or 1 (i.e., if the given descriptor is ready for I/O or not)
`getdtablesize()`	Gets the size of PPFDT (i.e., the maximum number of file descriptors that the system may use simultaneously)
`gethostbyname()`	Returns a pointer to a variable of the following structure describing an Internet host referenced by name
`gethostbyaddr()`	Returns a pointer to a variable of the following structure describing an Internet host referenced by address
`getservent()`	Returns a pointer to a variable of the `servent` structure with fields containing corresponding values in a line in the **/etc/services** file
`inet_addr()`	Converts and returns the specified string for an IP address in DDN to a 32-bit unsigned binary value in network byte order
`inet_aton()`	Converts the string containing an IPv4 address in DDN to an address in network byte order and stores it in an address structure
`inet_ntoa()`	Converts an IPv4 address in network byte order to a string containing the address in DDN
`inet_ntop()`	This is the newer version of the `inet_ntop()` function that works with both IPv4 and IPv6
`inet_pton()`	This is the newer version of the `inet_aton()` function that works with both IPv4 and IPv6

* These functions are normally used to convert IP addresses and port numbers returned by the `gethostbyname()` and `getservent()` library calls. On machines that have the same byte order as the network byte order, these functions are defined as null macros.

These macros, defined in **/usr/include/select.h, are normally used with the `select()` system call and are meant to set and test bits in a *descriptor set*, that is, a *bit mask* in which a particular number represents the state of the descriptor with that number. For example, the value in bit number 2 represents whether descriptor 2 is ready for I/O or not; the bit value 0 means "no" and the value 1 means "yes." The behavior of these macros is undefined if a descriptor value is negative or greater than the largest descriptor value on the system.

TABLE 20.3 Important Data Structures for Network-Based IPC and Their Purpose

Data Structure	Purpose
`struct sockaddr`	Most general structure for specifying the address of a socket
`struct sockaddr_un`	Structure for specifying the address of a UNIX domain socket
`struct in_addr`	Structure to store the IPv4 address
`struct sockaddr_in`	Structure used to specify the address of an Internet domain (`PF_INET` or `PF_INET6`) socket
`struct hostent`	Structure to maintain information about a host on the Internet
`struct servent`	Structure to maintain information about an Internet service

TABLE 20.4 Brief Description of Important Header Files

Header File	Purpose
`<netdb.h>`*	Definitions of various symbolic constants, data structures, and prototypes of functions to maintain and manipulate information about hosts, networks, servers, protocols, and Internet addresses
`<time.h>`	Definitions of various symbolic constants, data structures, and prototypes of functions to maintain and manipulate information about time
`<netinet/in.h>`	Definitions of assigned numbers according to RFC 1700 related to protocols, TCP ports, multicast addresses, and so on as symbolic constants, storage order for multibyte values, data structures (such as Internet style socket address structure), and prototypes of functions to maintain and manipulate them
`<arpa/inet.h>`	Definitions of various symbolic constants, data structures, and prototypes of functions to maintain and manipulate information about IP addresses for IPv4 and IPv6
`<sys/errno.h>`	Definitions of various symbolic constants for the errors produced by system and library calls
`<sys/select.h>`	Definitions of symbolic constants, data structures, macros, and prototypes for the `select()` and `pselect()` system calls
`<sys/socket.h>`	Definitions of symbolic constants, related data structures, and prototypes for system and library calls for socket types, address families, protocol families, message headers, addresses, and options
`<sys/stat.h>`	Definitions of symbolic constants for different statistics and values for files (inode number, hard link count, user ID, group ID, file size, time last accessed, time last modified, permission masks, type masks, etc.), macros to determine the type of a file, and prototypes for system calls to create and manipulate different types of files
`<sys/types.h>`	Definitions of symbolic constants for different types for items such as data units, inodes, file flags, disk addresses, Internet addresses, group IDs, process IDs, user IDs, thread IDs, access permissions, link counts, file offsets, resource limits, and so on; macros and function prototypes for related system calls
`<sys/un.h>`	Definitions of symbolic constants and data structure of socket addresses for UNIX-based IPC

* <abc.h> means /usr/include/abc.h

shows that our system uses an AMD64 processor. On Solaris, this command incorrectly shows the machine to be `i386`.

```
% cat byteorder.c
#include <stdio.h>

int main()
{
    int n;
    char *cp;

    n = 0x12345678;
    cp = (char*)(&n);
    if (*cp != 0x78)
       printf("Big Endian\n");
    else
```

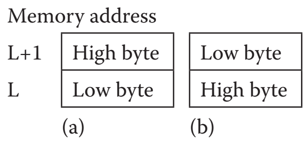


FIGURE 20.1 Storage orders for 16-bit data: (a) little endian (b) big endian.

```
       printf("Little Endian\n");
    return 0;
}
% gcc46 byteorder.c -w -o byteorder
% ./byteorder
Little Endian
% uname -p
amd64
%
```

20.4 THE CLIENT–SERVER MODEL

The design of applications to solve certain types of problems necessitates that we divide these applications into two independently running processes that communicate with each other to provide solutions for such problems. All Internet services are provided through such applications, including Web browsing, remote file transfer, remote login, remote program execution, video streaming, e-mail, and Internet games. Today, IPC primarily deals with processes of such applications communicating by sending messages back and forth. The Internet works on the *client–server* model. Web servers, database servers, and social media all use these types of interactions among client and server processes. In this model of communication, a process, called the *server*, offers some kind of service to other processes and runs on a host whose address is known. Another process, called the *client*, runs on the same host on which the server process runs, or on a different host, and initiates communication with the server process to use its service. We describe various client–server models throughout the rest of the chapter and different types of servers in detail in Section 20.7 and subsequent sections.

Server processes run forever and quietly wait for service requests from client processes. On receiving a request from a client process, the server process prepares a response, sends the response to the client process, and waits for the next request. Such servers are known as *iterative servers*. It is possible that, on receiving a client request, the server process creates a child process, delegates the rest of the communication with the client to that child process, and goes back to wait for another client request. Such servers are known as *concurrent servers*. In this style of client–server model, the original server process is known as the *master server* and the processes that it creates to handle communication with client processes are known as *slave processes*. Servers that need to respond to clients with one-time, short responses are usually iterative and those that need to interact with clients in request–response sessions are concurrent. Figure 20.2 shows the conceptual models for the iterative

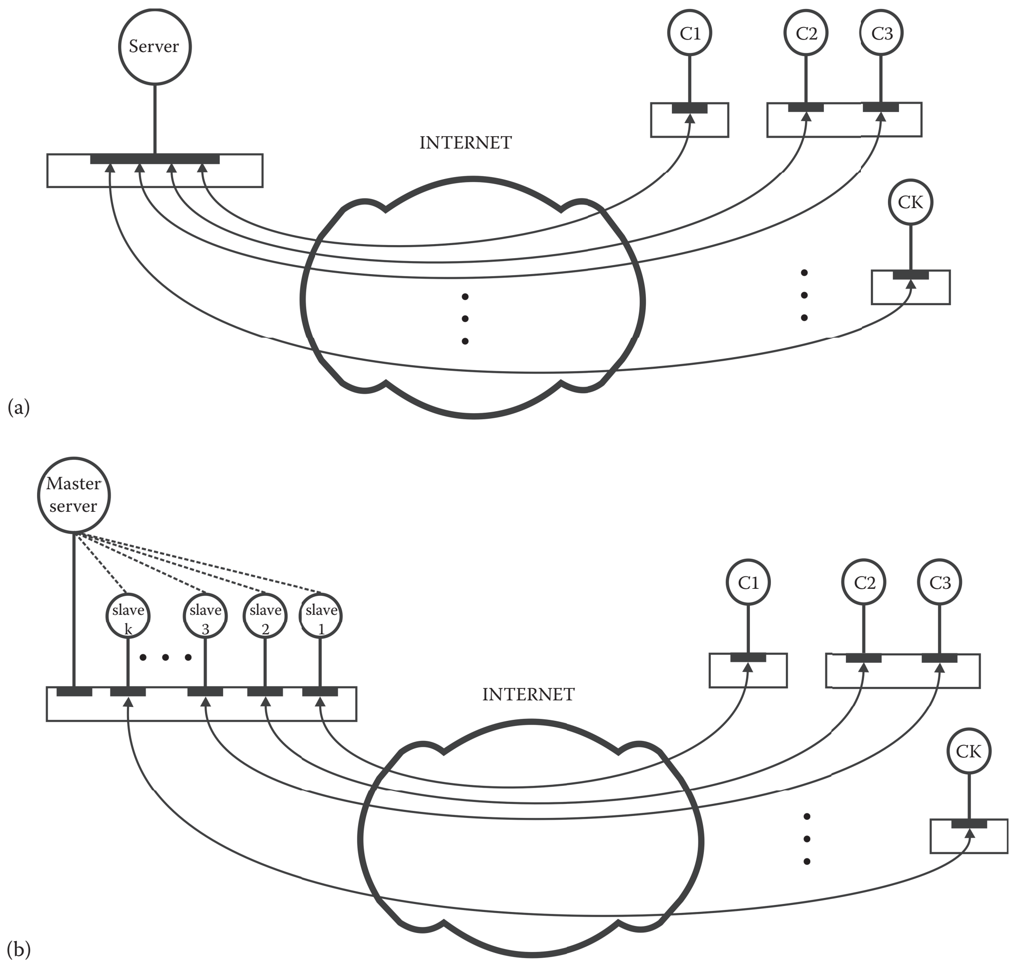


FIGURE 20.2 Conceptual models for (a) iterative server (b) concurrent server.

and concurrent servers with a server process serving *k* clients. Note that clients may run on a single host, including the host that runs the server process, or on multiple hosts.

Applications may be standard—also known as *well-known applications*—or nonstandard (*unknown*). Nontechnically speaking, well-known applications are those that most users know of and use, such as Web browsing, downloading files, remote execution of programs, voice or video streaming, e-mail, and logging on to a remote host. Technically speaking, well-known applications are those that are built around communication protocols described in Requests for Comments (RFCs) such as Hypertext Transfer Protocol (also called HTTP and WWW), File Transfer Protocol (FTP), Simple Mail Transfer Protocol (SMTP), and secure shell (SSH). Nonstandard applications may be sample applications written by students in a class or applications written for private use by individuals, groups of individuals, or employees of an organization or group of organizations.

20.4.1 Simplest Form of Communication

As discussed in detail in Chapter 19, the most primitive form of communication between UNIX processes is between a parent process and its children processes. This communication involves the use of the `exit()` system call (or the `exit()` library call) in a child process and the `wait()` system call (or a variant) in the parent process. Through this communication, a child communicates its exit status to the parent process. The exit status of the child is transferred from the `exit()` system call in the child to the `wait()` system call in the parent process. The UNIX kernel handles this communication implicitly and no explicit communication channel is involved.

20.4.2 Communication via Pipes

A pipe is a full-duplex (two-way) communication channel that allows two related processes to communicate with each other in terms of a stream of bytes, without message boundaries. It is normally used for communication between siblings, or parent and child. Although a pipe is a bidirectional communication channel, it is normally used for simplex (i.e., one-way) communication between two processes—a *reader process* and a *writer process*. Consequently, for two-way communication between two processes, a minimum of two pipes are usually required.

A pipe may be used to connect the standard output of a process (command) to the standard input of another process (command), as discussed in Chapter 9. The command-level syntax for accomplishing this task is `cmd1 | cmd2`, as in `sort datafile | grep "John Doe"`.

From an implementation point of view, a pipe is a *fixed-size main memory circular buffer* created and maintained by the UNIX kernel. In the operating system lexicon, it is also known as a *bounded buffer*. Communication using a pipe is, therefore, an implementation of the *bounded-buffer reader–writer problem*. The UNIX kernel handles the synchronization required for making the reader process wait when the pipe is empty and the writer process wait when the pipe is full.

The `pipe()` and `pipe2()` system calls may be used to create a pipe. Here is a brief description of the two calls.

```
#include <unistd.h>
int pipe(int filedes[2]);
int pipe2(int filedes[2], int flags);
```

Success: 0
Failure: -1 and kernel variable `errno` set to indicate the type of error

A call to `pipe()` or `pipe2()` returns two descriptors in the `filedes[2]` array, both allocated in the *per-process file descriptor table* (PPFDT) of the caller process. Thus, creating a pipe is equivalent to opening two files. However, unlike an open file, no file pointer is associated with a pipe. Thus, each write request to a pipe appends data to the current end of the pipe. The first descriptor, `filedes[0]`, is used to read from the pipe, and the second, `filedes[1]`, is used to write to the pipe.

TABLE 20.5 The Flags for the flags Argument in the `pipe2()` System Call

Flag	Meaning
`O_CLOEXEC`	Set the "close-on-exec" flag on the pipe descriptors. This means when a process executes an `exec()` system call, it does not inherit an already open pipe
`O_NONBLOCK`	Set pipe descriptors for nonblocking I/O. This means that a `read()` system call will not block when the pipe is empty and a `write()` system call will not block when the pipe is full

The `flags` argument in `pipe2()` is used to control the attributes of the pipe descriptors. Table 20.5 describes the possible flag values.

A bitwise-OR of these values may also be used for the `flags` argument. When the `flags` argument has a value of 0, the `pipe2()` system call behaves like the `pipe()` system call.

The maximum amount of data that can reside in a pipe is dictated by the size of the pipe as a bounded buffer (i.e., the fixed-size array of characters). When a pipe is full and a writer process wants to put data into it, the writer process is blocked by the UNIX kernel. Similarly, if a pipe is empty and a reader process tries to read data from the pipe, the reader process blocks. This is known as *blocking I/O* through a pipe.

The amount of data a writer process can write without interruption is known as the size of an *atomic write* into the pipe. It is defined as `PIPE _ BUF` in the **/usr/include/sys/limits.h** file under Solaris and in the **/usr/include/sys/syslimits.h** file under BSD. The value of `PIPE _ BUF` is 5120 bytes under Solaris and 512 bytes under BSD. Under Solaris, you can also use the `man limits` command to see the meaning of each symbolic constant described in the **limits.h** file, including `PIPE _ BUF`. Under BSD, you can use the `limits` command to display values of some of the symbolic constants defined in the **limits.h** file.

If multiple processes write to a pipe, then requests of `PIPE _ BUF` bytes or less are written atomically for each process. A request to write data greater than `PIPE _ BUF` bytes may have data interleaved, on arbitrary boundaries, with writes of other processes.

If the `O _ NONBLOCK` flag is not set, a request to write *n* bytes of data may block a thread if there is not enough space in the pipe to write the requested data. However, on successful completion, the `write()` system call returns *n*.

If the `O _ NONBLOCK` flag is set, then a `write()` system call never blocks and the thread continues its execution. When sufficient space is available in a pipe, a request to write `PIPE _ BUF` bytes or less completes successfully. If sufficient space is not available, the `write()` system call returns `–1` and `errno` is set to EAGAIN. If no space is available in a pipe, then a `write()` system call for writing more than `PIPE _ BUF` bytes returns `–1` with `errno` set to EAGAIN. If space is available for at least one byte, the call writes what it can and returns the number of bytes written. When the data previously written to the pipe has been read, the call writes at least `PIPE _ BUF` bytes.

The `pipe()` and `pipe2()` calls may fail for the reasons listed in Table 20.6.

TABLE 20.6 Reasons for the `pipe()` and `pipe2()` System Calls to Fail

Reason for Failure	**Value of** `errno`
The PPFDT does not have two unused file descriptors	EMFILE
The system file table is full	EMFILE
The kernel does not have enough memory to create a pipe	ENOMEM
The `flags` argument in `pipe2()` is not valid	EINVAL

20.4.2.1 Allocation of Pipe Descriptors

The `create _ pipe.c` program shown next creates a pipe, and displays the values of descriptors for the read and write ends of the pipe. The UNIX kernel opens three standard files automatically for each process using file descriptors 0 (standard input), 1 (standard output), and 2 (standard error). Therefore, when we run this program, the kernel allocates the next two unused file descriptors, 3 and 4, to the read and write ends of the pipe, respectively. Figure 20.3 shows the pipe with its relationship to the PPFDT.

```
% cat create_pipe.c
#include <unistd.h>

int main(void)
{
       int data_channel[2];

       if (pipe(data_channel) == -1) {
           perror("Pipe failed");
           exit(1);
       }
       printf("The pipe descriptors are: \n");
       printf("   Read end:  %d\n", data_channel[0]);
       printf("   Write end: %d\n", data_channel[1]);
       exit(0);
}
```

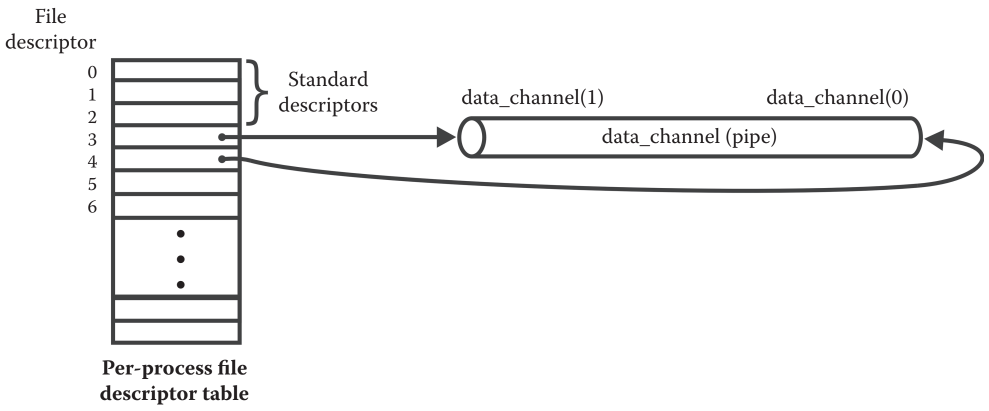


FIGURE 20.3 Pipe created by the `create _ pipe.c` program and its relationship with PPFDT.

```
% gcc46 create_pipe.c -w -o createpipe
% ./createpipe
The pipe descriptors are:
       Read end:  3
       Write end: 4
%
```

20.4.2.2 One-Way Communication

The `pipe _ talk.c` program creates a pipe and spawns a child process. The child process, among other things, as discussed in Chapter 19, inherits the parent's PPFDT. The child process closes the write end of the pipe and reads data from the read end of the pipe. The child process blocks if there is nothing in the pipe. The parent process, on the other hand, closes the read end of the pipe and writes data to the child process using the write end of the pipe. The child process displays on the screen whatever it reads from the pipe. Figure 20.4 shows the setup for the program in a pictorial form.

```
% cat pipe_talk.c
#include <unistd.h>

#define SIZE 32

const char *Child_Greeting="Hello, mom!\n";

int main(void)
{
        int data_channel[2], pid, nr, nw, nbytes;
        char buf[SIZE];

        if (pipe(data_channel) == -1) {
            perror("Pipe failed");
            exit(1);
        }
        pid = fork();
        if (pid == -1) {
```

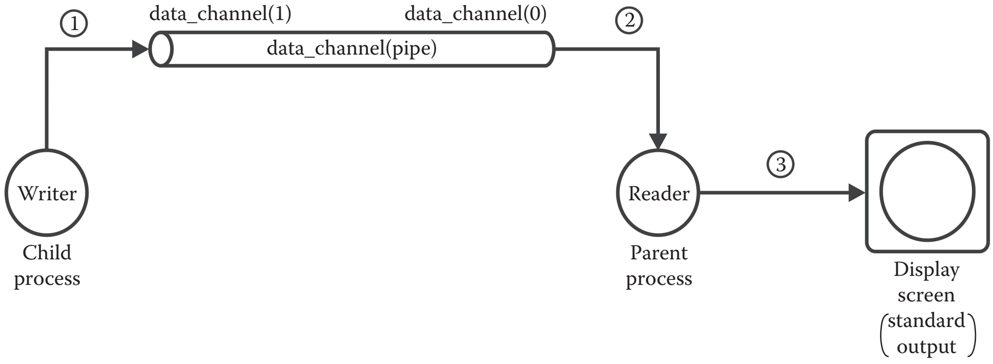


FIGURE 20.4 Pictorial representation of the IPC setup in `pipe _ talk.c`

```
            perror("Fork failed");
            exit(1);
        }
        nbytes = strlen(Child_Greeting);
        if (pid == 0) {
            close(data_channel[0]);
            nw = write(data_channel[1], Child_Greeting, nbytes);
            if (nw == -1) {
               perror("Write error");
               exit(1);
            }
            exit(0);
        }
        /* Parent process */
        close(data_channel[1]);
        nr = read(data_channel[0], buf, nbytes);
        if (nr == -1) {
            perror("Read error");
            exit(1);
        }
        nw = write(1, buf, nr);
        if (nw == -1) {
            perror("Write to stdout failed");
            exit(1);
        }
        printf("Well done, son!\n");
        exit(0);
}
% gcc46 pipe_talk.c -w -o pipetalk
% ./pipetalk
Hello, mom!
Well done, son!
%
```

20.4.2.3 Two-Way Communication

We now discuss a program in which the reader and writer processes carry out two-way communication using two pipes. The `pipe _ 2way _ talk.c` program opens two pipes, `pipe1` and `pipe2`, and creates a child process. The child process sends a message to the parent process using `pipe1` and reads the parent's response from `pipe2`. The parent does it the other way round. Whatever data the parent and child processes read from their respective pipes, they throw to standard output. Figure 20.5 shows the setup in a pictorial form.

```
% cat pipe_2way_talk.c
#include <unistd.h>

#define SIZE 32
```

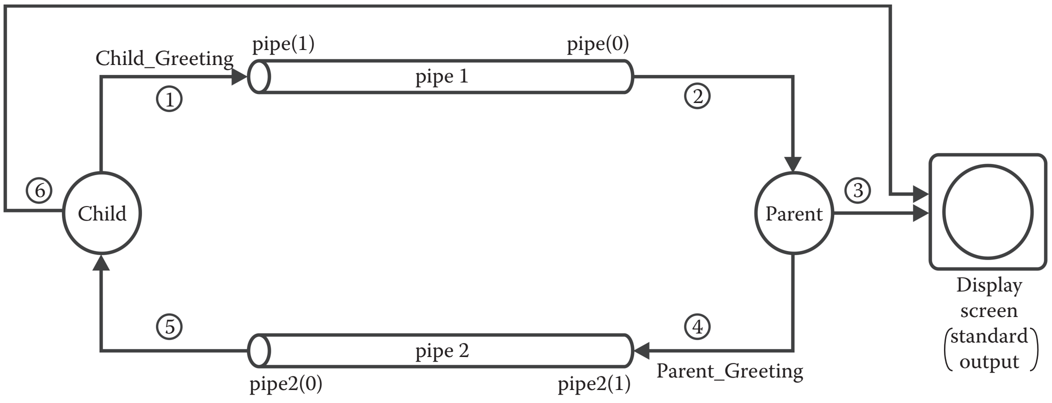


FIGURE 20.5 Pictorial representation of the IPC setup in `pipe_2way_talk.c`.

```
const char *Child_Greeting="Hello, mom!\n";
const char *Parent_Greeting="Well done, son!\n";

int main(void)
{
      int pipe1[2], pipe2[2];
      int pid, nr, nw, status, sizec, sizep;
      char buf[SIZE];

      sizec = strlen(Child_Greeting);
      sizep = strlen(Parent_Greeting);
      if (pipe(pipe1) == -1) {
          perror("Pipe1 failed");
          exit(1);
      }
      if (pipe(pipe2) == -1) {
          perror("Pipe2 failed");
          exit(1);
      }
      pid = fork();
      if (pid == -1) {
          perror("Fork failed");
          exit(1);
      }
      if (pid == 0) {
          close(pipe1[0]);
          close(pipe2[1]);
          nw = write(pipe1[1], Child_Greeting, sizec);
          if (nw == -1) {
            perror("Write to pipe1 error in child");
            exit(1);
          }
          nr = read(pipe2[0], buf, sizep);
          if (nr == -1) {
            perror("Read pipe2 error in child");
            exit(1);
          }
          nw = write(1, buf, nr);
          if (nw == -1) {
            perror("Write to stdout failed in child");
            exit(1);
          }
          close(pipe1[1]);
          close(pipe2[0]);
          exit(0);
      }
      /* Parent process */
      close(pipe1[1]);
      close(pipe2[0]);
      nr = read(pipe1[0], buf, sizec);
      if (nr == -1) {
          perror("Read pipe1 error in parent");
          exit(1);
      }
      nw = write(1, buf, nr);
      if (nw == -1) {
          perror("Write to stdout failed in parent");
          exit(1);
      }
      nw = write(pipe2[1], Parent_Greeting, sizep);
      if (nw == -1) {
          perror("Write to pipe2 error in parent");
          exit(1);
      }
      close(pipe1[0]);
      close(pipe2[1]);
      wait(&status);
      exit(0);
}
% gcc46 pipe_2way_talk.c -w -o p2wt
% ./p2wt
Hello, mom!
Well done, son!
%
```

20.4.2.4 Widowed Pipe

A pipe that has one end closed is called a *widowed pipe*. When a writer process writes to a widowed pipe, the UNIX kernel sends the SIGPIPE signal to the writer process that results in the termination of the writer process. The default action on this signal is that the kernel terminates the process and displays the "`Broken pipe`" message. When a reader process reads from a widowed pipe, it receives an end-of-file (eof) message after the reader has read all of the data in the pipe. This means that a `read()` system call on a widowed pipe returns the value 0.

In the `broken _ pipe.c` program shown, we demonstrate how a widowed pipe may be created. The program creates a pipe, displays the read end and write end descriptors of the pipe, closes the read end of the pipe (i.e., no reader can read from the pipe), and writes a greeting message using the write end of the pipe. We compile the program to generate the executable code in the **brokenp** file. The execution of the program works perfectly until it tries to write to the pipe after closing its read end. The `write()` system call generates the SIGPIPE signal and, due to the kernel's default action, the "`Broken pipe`" message is displayed before the program terminates.

```
% cat broken_pipe.c
#include <unistd.h>

int main(void)
{
        int data_channel[2], nw;

        if (pipe(data_channel) == -1) {
            perror("Pipe failed");
            exit(1);
        }
        printf("The pipe descriptors are: \n");
        printf("        Read end:  %d\n", data_channel[0]);
        printf("        Write end: %d\n", data_channel[1]);
        close(data_channel[0]);
        nw = write(data_channel[1], "Hello, world!\n", 14);
        printf("This and subsequent statements are never
        executed.\n");
        exit(0);
}
% gcc46 broken_pipe.c -w -o brokenp
% ./brokenp
The pipe descriptors are:
        Read end:  3
        Write end: 4
Broken pipe
%
```

Note that the compilation and execution of the program was done on a BSD system. On Solaris, the program compiles and works in the same way as on the BSD system, except that it terminates the process without displaying the "`Broken pipe`" error message. You can use the `signal()` call to intercept the SIGPIPE signal, display the desired error message, and terminate the program. You will be asked to do this in one of the problems at the end of this chapter.

EXERCISE 20.1

Compile and run the `create _ pipe.c`, `pipe _ talk.c`, `pipe _ 2way _ talk.c`, and `broken _ pipe.c` programs in the previous session to make sure they work on your system as expected.

20.5 COMMUNICATION BETWEEN UNRELATED PROCESSES ON THE SAME COMPUTER

Two or more related or unrelated processes on the same machine can communicate with each other using several UNIX IPC channels, including a named pipe (also known as a FIFO) and a socket. We discuss FIFOs in this section and sockets in Section 20.6. Earlier, we discussed FIFOs in Section 9.15. Our treatment of the topic was focused on the command line use of FIFOs for connecting shell commands with each other to perform complex tasks that cannot be performed by existing commands. Here, we discuss the details of the underlying structure of a FIFO as an IPC channel and the UNIX application programmer's interface (API) that allows creation of FIFOs and their use.

As stated earlier, a FIFO is a named pipe. It is a pipe that has a name in the file system name space, an associated file type, related kernel data structures that contain its attributes, and the main memory bounded-buffer that contains data in transition through the pipe. The pipe part of the FIFO, a main memory buffer, is created when a process opens the FIFO and is destroyed when the process closes the FIFO. Thus, unlike a pipe that is purely a main memory object and is *process persistent*, a FIFO is an amalgamation of disk and memory objects. The pipe part of it is process persistent and the name part is *file system persistent*. To sum up, when you create a FIFO, the kernel creates a pipe in main memory and connects it with a file system through a pathname in the file system name space and associated resources including an inode. This allows you to access a FIFO as a file system object.

To use a FIFO as an IPC channel, you create it with a pathname and then open it for reading, writing, or both. However, as expected, just like a pipe, a FIFO also does not have a file pointer associated with it. New data is written at the current end of the FIFO and existing data is read from the front of the FIFO.

When you create a FIFO, it does not contain anything, in the same way as an empty regular file; and, depending on the UNIX system you use, its disk usage is zero or one block. When you write data to a regular file, its disk usage depends on the amount of data written to the file with a minimum usage of five blocks on BSD and nine blocks on Solaris. This amount may be different for other UNIX systems. However, the disk usage for a FIFO does not change whether it is empty or full, because its data is maintained in the memory-resident pipe.

The following session was captured on a BSD machine. We create a FIFO, called **fifo1**, and an empty file, called **greeting**. Both are allocated inodes numbered 4933 and 4934. The output of the `ls -l fifo1 greeting` command shows that both are empty. The output of the `du fifo1 greeting` command shows that both use one disk block. We put "`Hello, world!`" (14 bytes) in the **greeting** file as well as **fifo1**. The output of the second `ls -l fifo1 greeting` command shows that **fifo1** is still empty, but **greeting** contains 14 bytes. The output of the `du fifo1 greeting` command shows that whereas **fifo1** still uses one disk block, the **greeting** file uses five disk blocks. Even after we empty **fifo1** using the `cat fifo1` command, the disk usage of **fifo1** remains unchanged. This session shows that when we put data into **fifo1**, the data does not go into a disk object associated with it but into the memory buffer (pipe) associated with **fifo1**.

```
% mkfifo fifo1
% touch greeting
% ls -i fifo1 greeting
4933 fifo1    4934 greeting
% ls -l fifo1 greeting
prw-r--r--  1 sarwar  faculty  0 Apr 18 17:29 fifo1
-rw-r--r--  1 sarwar  faculty  0 Apr 18 17:29 greeting
% du fifo1 greeting
1       fifo1
1       greeting
% cat > greeting
Hello, world!
% cat greeting > fifo1 &
[1] 68928
% ls -l fifo1 greeting
prw-r--r--  1 sarwar  faculty   0 Apr 18 17:29 fifo1
-rw-r--r--  1 sarwar  faculty  14 Apr 18 17:31 greeting
% du fifo1 greeting
1       fifo1
5       greeting
% cat fifo1
Hello, world!
[1]  + Done                          cat greeting > fifo1
% du fifo1 greeting
1       fifo1
5       greeting
%
```

The following session was captured on a Solaris machine to show that FIFOs work the same way on Solaris too. The only difference is that, whereas on a BSD system an empty FIFO uses one disk block, it does not use any disk space on Solaris, as shown in the output of the `du fifo1 greeting` command.

```
$ mkfifo fifo1
$ touch greeting
$ ls -i fifo1 greeting
```

```
       318 fifo1            319 greeting
$ ls -l fifo1 greeting
prw-r--r--   1 sarwar   faculty        0 Apr 18 17:42 fifo1
-rw-r--r--   1 sarwar   faculty        0 Apr 18 17:42 greeting
$ du fifo1 greeting
0       fifo1
1       greeting
$ cat > greeting
Hello, world!
$ cat greeting > fifo1 &
[1] 10044
$ ls -l fifo1 greeting
prw-r--r--   1 sarwar   faculty        0 Apr 18 17:42 fifo1
-rw-r--r--   1 sarwar   faculty       14 Apr 18 17:43 greeting
$ du fifo1 greeting
0       fifo1
9       greeting
$ cat fifo1
Hello, world!
[1]+  Done                  cat greeting > fifo1
$ du fifo1 greeting
0       fifo1
9       greeting
$
```

EXERCISE 20.2

Replicate these shell sessions on your UNIX system(s). Do they produce the same results? If not, list the differences between the results of our sessions and your sessions.

You can use any of the `mknod()`, `mknodat()`, `mkfifo()`, or `mkfifoat()` system calls to create a FIFO. However, the `mknod()` and `mknodat()`system calls require superuser privileges. The primary purpose of these system calls is to create special files, but they can also be used to create FIFOs. The `mkfifo()` and `mkfifoat()` calls do eventually invoke the `mknod()` system call. Once a FIFO has been created, you can use the `open()`, `read()`, `write()`, and `close()` system calls to perform I/O with it through multiple reader and writer processes. The last process that uses a FIFO, usually a reader process, closes the FIFO and removes it from the file system using the `unlink()` system call. Because multiple processes can write to a FIFO, the UNIX kernel ensures that data up to `PIPE _ BUF` bytes written by multiple processes each is written atomically and does not interleave. `PIPE _ BUF` is defined in **/usr/include/sys/param.h** to be 5120 bytes on Solaris and in **/usr/include/sys/syslimits.h** to be 512 bytes on BSD.

We first describe the `mkfifo()` and `mkfifoat()` system calls, listed as library calls under Solaris, and their manual pages may be viewed by using the `man –s 3c mkfifo` command. We then discuss a few sample programs to describe the use of FIFOs as IPC channels. We primarily use the `mkfifo()` call in our sample programs. Here are brief descriptions of the `mkfifo()` and `mkfifoat()` system calls.

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *path, mode_t mode);
int mkfifoat(int fd, const char *path, mode_t mode);
```

Success: `0`
Failure: `-1` and kernel variable `errno` set to indicate the type of error

The `mkfifo()` creates a FIFO with the name given in the `path` argument having access permissions specified in the `mode` argument. As is the case with other types of files, access permissions are restricted by the current `umask`. In the `mkfifoat()` system call, the `path` argument is relative to the directory associated with the file descriptor `fd` and not the current working directory of the user. If the `fd` argument is `AT _ FDCWD`, the behavior of the `mkfifoat()` call is identical to the `mkfifo()` call.

The `mkfifo()` and `mkfifoat()` calls may fail mostly for the same reasons that the `creat()` and `open()` system calls may fail, as discussed in Section 18.8.1 and listed in Table 18.4. A few additional reasons for the failure of these calls are listed in Table 20.7.

The behavior of a FIFO that is not fully opened for I/O is similar to that of a pipe under the same condition. A write to a FIFO that no process has opened for reading results in a SIGPIPE signal to the writer process. When the last process to write to a FIFO closes it, an eof is sent to the reader process.

We now discuss a sample client–server model to illustrate the use of FIFOs for IPC between unrelated processes on the same machine. Figure 20.6 shows the algorithms for the client and server processes.

Here are the **fifo.h**, **client.c**, and **server.c** files that implement this client–server model.

```
% cat fifo.h
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/errno.h>
```

TABLE 20.7 Reasons for the `mkfifo()` and `mkfifoat()` System Calls to Fail

Reason for Failure	**Value of `errno`**
The directory under which the FIFO is to be created does not have write permission, or a component directory in the path prefix is not searchable	`EACCES`
The FIFO named in `path` exists	`EEXIST`
For the `mkfifoat()` call, the `fd` argument is neither `AT_FDCWD` nor a valid descriptor for searching and the `path` argument is not an absolute pathname	`EBADF`
For the `mkfifoat()` call, the `fd` argument is neither `AT_FDCWD` nor a valid descriptor for a directory and the `path` argument is not an absolute pathname	`ENOTDIR`

Server process	**Client process**
1. Create two **FIFOs**, **FIFO1** and **FIFO2**	1. Open **FIFO1** for writing and **FIFO2** for reading
2. Open **FIFO1** for reading and **FIFO2** for writing	2. Write "Hello, world!" to server through **FIFO1**
3. Read "Hello, world!" from client through **FIFO1**	3. Read **FIFO2** for a message from server
4. Display this message on the screen by writing it to standard output	4. Display this message, "Hello, class!", on the screen by writing it to standard output
5. Write "Hello, class!" to client through **FIFO2**	5. Close **FIFO1** and **FIFO2**
6. Close **FIFO1** and **FIFO2**	6. Remove **FIFO1** and **FIFO2** from the file system
7. Exit	7. Exit

FIGURE 20.6 Algorithms for the client and server processes.

```
extern int      errno;

#define FIFO1   "/tmp/fifo.1"
#define FIFO2   "/tmp/fifo.2"
#define PERMS   0666
#define SIZE    512

static char* message1 = "Hello, world!\n";
static char* message2 = "Hello, class!\n";
% cat client.c
#include "fifo.h"

int main(void)
{
    char buff[SIZE];
    int readfd, writefd;
    int n, size;

    /* Open FIFOs. Assume that the server
       has already created them. */
    if ((writefd = open(FIFO1, 1)) == -1) {
        perror ("client open FIFO1");
        exit (1);
    }
    if ((readfd = open(FIFO2, 0)) == -1) {
        perror ("client open FIFO2");
        exit (1);
    }

    /* client (readfd, writefd); */
    size = strlen(message1);
    if (write(writefd, message1, size) != size) {
        perror ("client write1");
        exit (1);
    }
    if ((n = read(readfd, buff, size)) == -1) {
        perror ("client read");
        exit (1);
    }
    else
        if (write(1, buff, n) != n) {
            perror ("client write2");
            exit (1);
        }
    close(readfd);
    close(writefd);

    /* Remove FIFOs now that we are done using them */
    if (unlink (FIFO1) == -1) {
        perror("client unlink FIFO1");
        exit (1);
    }
    if (unlink (FIFO2) == -1) {
        perror("client unlink FIFO2");
        exit (1);
    }
    exit (0);
}
% cat server.c
#include "fifo.h"

int main(void)
{
    char buff[SIZE];
    int readfd, writefd;
    int n, size;

    /* Create two FIFOs and open them for
       reading and writing */
    if ((mknod (FIFO1, S_IFIFO | PERMS, 0) == -1)
                && (errno != EEXIST)) {
        perror ("mknod FIFO1");
        exit (1);
    }
    if (mkfifo(FIFO2, PERMS) == -1) {
        unlink (FIFO1);
        perror("mknod FIFO2");
        exit (1);
    }
    if ((readfd = open(FIFO1, 0)) == -1) {
        perror ("open FIFO1");
        exit (1);
    }
    if ((writefd = open(FIFO2, 1)) == -1) {
        perror ("open FIFO2");
        exit (1);
    }

    /* server (readfd, writefd); */
    size = strlen(message1);
    if ((n = read(readfd, buff, size)) == -1) {
        perror ("server read");
        exit (1);
    }
    if (write (1, buff, n) < n) {
        perror("server write1");
        exit (1);
    }
    size = strlen(message2);
    if (write (writefd, message2, size) != size) {
        perror ("server write2");
        exit (1);
    }
    close (readfd);
    close (writefd);
}
%
```

Figure 20.7 shows the compilation and execution of the client–server model by using two terminal windows on our system. We compile the `client.c` and `server.c` programs and save the executable codes in the client and server files, respectively. We then run the server program in Window 1, followed by running the client program in Window 2. The client process sends the "`Hello, world!`" message to the server process through **FIFO1** and then waits for a message from the server process. The server process receives the client message and displays it on standard output. It then sends the "`Hello, class!`" message to the client process through **FIFO2**, closes both FIFOs, and exits. The client process receives the message and displays it on the screen. It then closes both FIFOs and removes them from the file system. Figure 20.8 shows the pictorial representation of the running of the client–server model.

The client–server model discussed is for communication between two processes running on the same machine. In this case, the communication is an exchange of one message each from client to server and vice versa. The model may be extended to the exchange of multiple messages between the two processes. You can extend this model such that the server process may serve multiple clients. Figure 20.9 shows the general view of such a model.

Terminal window 1

```
% gcc46 server.c -w -o server
% ./server
Hello, world!
%
```

Terminal window 2

```
% gcc46 client.c -w -o client
% ./client
Hello, class!
%
```

FIGURE 20.7 Compilation and running of the client–server model in two terminal windows.

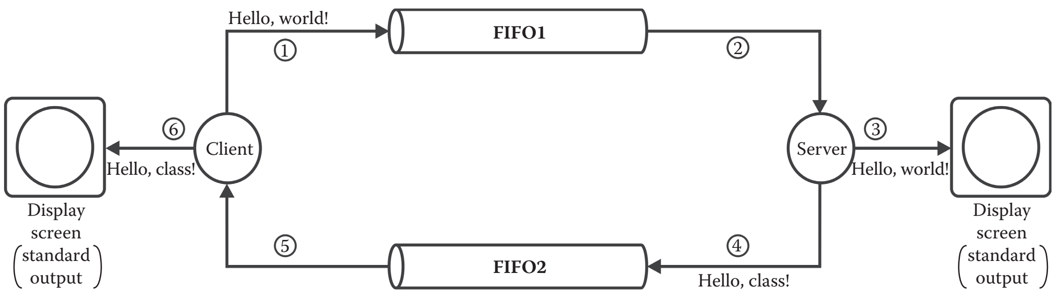


FIGURE 20.8 Pictorial representation of the execution of the client–server model using FIFOs.

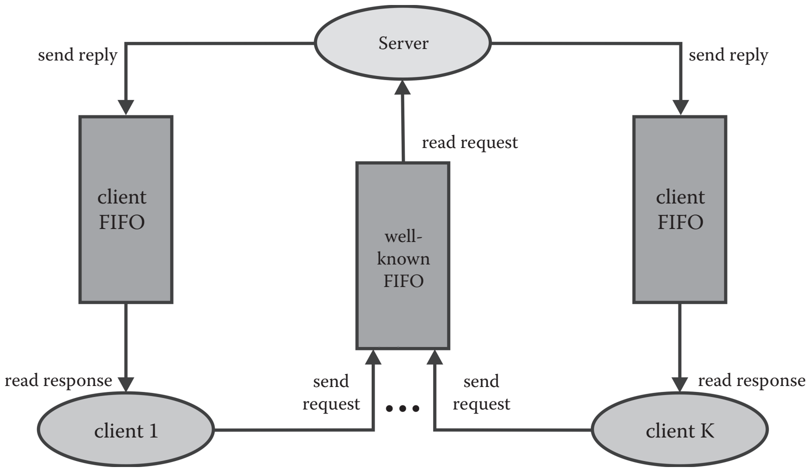


FIGURE 20.9 Client–server model with a server and multiple simultaneous clients using FIFOs.

EXERCISE 20.3

Compile and run this client–server model on your UNIX system. Does it work as expected? If not, identify and list reasons for the issues.

EXERCISE 20.4

Are `mkfifo()` and `mkfifoat()` are available on your UNIX system as system calls or library calls?

EXERCISE 20.5

What is the value of `PIPE _ BUF` on your UNIX system? Which file contains it? What does this value indicate?

The basic algorithms for the client and server processes for this client–server model are shown in Figure 20.10. Note that Steps 3–5 in the server code are typically implemented as an infinite loop.

Server process	Client process
1. Create a "well-known" FIFO, i.e., create a FIFO and make its pathname well known 2. Open the well-known FIFO for reading 3. Read a client request via the well-known FIFO; as part of its request a client sends the pathname of its FIFO to server 4. Prepare response and send it to client FIFO 5. Go to Step 3	1. Create a FIFO, called "client" FIFO, for reading server response(s) 2. Open server's well-known FIFO for writing 3. Prepare a request for the server that contains the pathname of client FIFO 4. Send request to server via server's well-known FIFO 5. Receive server's response via client FIFO 6. Response to last request? No, go to 3. 7. Close well-known and client FIFOs 8. Remove client FIFO from the file system 9. Exit

FIGURE 20.10 Algorithms for the client and server processes.

20.6 COMMUNICATION BETWEEN UNRELATED PROCESSES ON DIFFERENT COMPUTERS

UNIX provides several IPC channels for communication between unrelated processes on the same or different machines on a network. The most commonly used channel is the *socket*. In this section, we discuss what a socket is and how it may be used for communication between processes. The communication between processes may be connection oriented or connectionless. We also explore the software architecture of the client–server models for the various Internet services by using the socket as the communication channel between the client and server processes. We will code some of the basic models and explain the source codes. We leave implementation of other models as end of chapter problems.

20.6.1 Socket-Based Communication

As stated in Chapter 4, a socket is a file type in UNIX. From the IPC point of view, a *socket* is a full-duplex IPC channel that may be used for communication between related or unrelated processes on the same or different machines. Both communicating processes need to create a socket to handle their side of communication; reading and writing. A socket is, therefore, called an *endpoint of communication*. It is the IPC channel of choice for network-based communication between processes under the client–server paradigm.

Like a FIFO, once a socket has been created, it is used by following the open–read–write–close paradigm used for typical file I/O. A socket remains in the system for as long as the process that creates it is up and running. Thus, a socket is *process persistent*.

20.6.2 Creating a Socket

You can create a socket by using the `socket()` system call. Here is a brief description of the `socket()` system call.

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

Success: Socket descriptor
Failure: -1 and kernel variable `errno` set to indicate the type of error

The call returns a file descriptor from the PPFDT, known as the *socket descriptor*. Before a socket may be used, it needs to be set for a particular type of communication: connectionless or connection oriented. The `domain` argument specifies the domain (i.e., protocol family) under which the communication between a pair of sockets will take place. The value of this argument selects the protocol family that would be used for communication between sockets. These families are defined in **/usr/include/sys/socket.h**. A socket of a particular domain may support different types of communication under different protocols. The `type` argument specifies the type of communication for which the socket would be used. Communication may only take place between a pair of sockets of the same type. The `protocol` argument specifies the protocol to be used for the socket type within a given protocol family. However, a particular protocol needs to be specified if multiple protocols exist to support the requested type of communication under the given domain. If you set the `protocol` argument to 0, the system chooses the correct type of protocol for the given type of communication under the selected domain.

Table 20.8 lists some of the commonly used socket domains and Table 20.9 shows the types of communication supported between a pair of sockets. The protocols for stream-oriented and datagram-oriented communication are TCP and User Datagram Protocol (UDP), respectively.

You use the `PF _ LOCAL` domain to create a socket for communication between processes on the same host using internal protocol(s). The `PF _ INET` and `PF _ NET6` domains are used for communication between processes on the Internet using the IPv4 and IPv6 protocols, respectively.

You can OR the flags given in Table 20.10 to mark a newly created socket as "close-on-exec" and/or set it for "nonblocking" I/O. When a socket is marked as "close-on-exec," it

TABLE 20.8 Commonly Used Socket Domains

Protocol Family	**Purpose**
`PF_LOCAL` (Old `PF_UNIX`)	Host-internal protocols
`PF_INET`	IPv4 protocols
`PF_INET6`	IPv6 protocols

TABLE 20.9 Commonly Used Types of Communication between a Pair of Sockets

Socket Type	**Communication Type**
`SOCK_STREAM`	Stream-oriented
`SOCK_DGRAM`	Datagram-oriented
`SOCK_RAW`	Datagram-oriented (only available to superuser)

TABLE 20.10 Flags for the `type` Field

Flag	**Purpose**
`SOCK_CLOEXEC`	Mark the descriptor as "close-on-exec"
`SOCK_NONBLOCK`	Set the descriptor "nonblocking" I/O

TABLE 20.11 A Few Reasons for `socket()` to Fail

Reason for Failure	Value of `errno`
Wrong `type` and/or `protocol` argument	`EACCES`
`domain` is not supported	`EAFNOSUPPORT`
PPFDT is full (i.e., no free file descriptor) or system-wide file table is full	`EMFILE`
Insufficient buffer space in kernel	`ENOBUFS`
Specified `protocol` not supported in the given domain	`EPROTONOSUPPORT`
Specified socket `type` not supported by the given protocol	`EPROTOTYPE`

is closed when the process overwrites itself with an executable code using a call from the `exec()` family such as `execlp()`. The `read()` and `write()` system calls do not block on an empty and full socket, respectively, if the socket is marked as "nonblocking."

The `socket()` system call may fail for various reasons. Some of the reasons for the failure of a `socket()` are listed in Table 20.11.

20.6.3 Domains of Socket-Based Communication

Socket-based IPC may take place in several domains, but two are most commonly used: *UNIX domain* and *Internet domain*. We briefly describe when these domains of communication are used. In the next section, we show how a socket may be created for communication under a particular domain.

20.6.3.1 UNIX Domain Socket (`PF _ LOCAL` or `PF _ UNIX`)

To create a socket for communication under the UNIX domain, `PF _ LOCAL` or `PF _ UNIX` is used as the `domain` argument in the `socket()` system call. The IPC under this domain is between processes on the same machine with the socket address specified as the pathname in the file system, in the same way as that for a FIFO. UNIX domain protocols are not an actual protocol suite but a technique used to perform client–server communication between processes on the same machine using the socket API.

The UNIX domain sockets are preferred over FIFOs and the Internet domain sockets for IPC between processes on the same machine for the following reasons:

- UNIX domain sockets are full duplex.
- UNIX domain sockets are twice as fast as TCP sockets. For this reason, they are used in the client–server model for X Window System as well as for communication between a client and server when both are on the same host. An example of the latter case is when a printer client running on a machine sends a print job to the print server that also runs on the same machine.
- UNIX domain sockets are used for passing descriptors between processes on the same host.

20.6.3.2 Internet Domain Socket (`PF _ INET` or `PF _ INET6`)

To create a socket for communication under the Internet domain, `PF _ INET` or `PF _ INET6` is used as the domain in the `socket()` system call. The IPC under this domain takes place between processes on the same machine or on different machines on a TCP/IP network, including the Internet. The address of an Internet domain socket is an IP address and a port number. The Internet domain protocols use the TCP/IP protocol suite for communication using the client–server paradigm.

The Internet domain sockets are the most commonly used channels for IPC and are the glue required for the construction of client–server applications for Internet services. In the next section, we discuss the most common types of communication that are carried out using sockets and the underlying protocols that support such communication.

20.6.4 Types of Communication Using a Socket

IPC using sockets may be *connection oriented* or *connectionless*. In the connection-oriented style of communication, the two communicating processes create sockets of the required type and have them connected before starting communication. The connection between the two sockets is established using the *three-way TCP handshake*. Thus, a *virtual connection* is established between the two sockets, and the sender and receiver processes communicate using the `write()` and `read()` system calls, respectively. The `SOCK _ STREAM` type of socket is used for connection-oriented, reliable, error-free, and in-sequence stream-oriented communication with no message/packet boundaries.

In the connectionless style of communication, the sender process simply sends messages to the receiver process and hopes that they will be delivered. The `SOCK _ DGRAM` type socket is used for connectionless, unreliable (no guarantees), and datagram-oriented communication. A *datagram* is a small, fixed-length packet/message. Connection-oriented communication is like the guaranteed delivery service provided by FedEx, UPS, or DHL, and connectionless communication is like the normal, best-effort delivery service provided by a country's mail service such as the U.S. Post Service.

20.6.4.1 Stream Socket (`SOCK _ STREAM`)

To create a socket for stream-oriented communication using IPv4, you specify `PF _ INET` as the domain and `SOCK _ STREAM` as the type of communication. TCP is the transport-level protocol that provides this type of communication. Thus, you can create an Internet domain socket for stream-type connection-oriented communication by using the following code snippet. Note that a value of 0 is used in the `protocol` argument to let the kernel choose the appropriate protocol for the requested style of communication.

```
int s; /* Socket descriptor */
s = socket(PF_INET, SOCK_STREAM, 0);
```

If you also want to mark this socket as nonblocking, you would use the following call:

```
int s;
s = socket(PF_INET, SOCK_STREAM | SOCK_ NONBLOCK, 0);
```

20.6.4.2 Datagram Socket (`SOCK _ DGRAM`)

To create an IPv4 socket for datagram-oriented communication, you would need to specify `PF _ INET` as the domain and `SOCK _ DGRAM` as the type of communication. UDP is the transport-level protocol that provides this type of communication. Thus, you can create an Internet domain socket for datagram-oriented, connectionless communication by using the following code snippet:

```
int s; /* Socket descriptor */
s = socket(PF_INET, SOCK_DGRAM, 0);
```

If you want to also mark this socket "close-on-exec," you would use the following call:

```
int s;
s = socket(PF_INET, SOCK_DGRAM | SOCK_CLOEXEC, 0);
```

20.6.4.3 Compiling and Running Programs on PC-BSD and Solaris

You can compile your programs using the `gcc46` (or `gcc48`) command on PC-BSD and `gcc` on Solaris. On PC-BSD, you can compile a source program by using the following command syntax:

```
gcc46 -w source.c -o binary
```

where `source.c` is the name of the C program file that contains the source code and `binary` is the name of the file that contains the executable code for the given source code. The `–w` switch is used to suppress warnings. On Solaris, programs that use network-related system calls, including `socket()`, `bind()`, `accept()`, and so on, require that the `gcc` compiler is run using the `–lsocket` switch. To compile a program that uses socket library calls such as `gethostbyname()`, you need to specify the `–lnsl` switch also. So, you use the following compiler command syntax to generate executable code for a server program on Solaris.

```
gcc -w -lsocket server.c -o server
```

In the client–server software that we develop in this chapter, we do not use the network library calls that require the `–lnsl` switch. However, we do use the network library calls in our client software. Thus, you use the following compiler command to generate executable code for client software on Solaris.

```
gcc -w -lsocket -lnsl client.c -o client
```

For all the programs that we discuss in the intervening sections, we show compilation and execution of the programs on our PC-BSD system.

The IPv4 addresses of our PC-BSD and Solaris machines are 202.147.169.196 and 202.147.169.197, respectively. Since we test run our server processes on these machines, we use these IP addresses as command line arguments for the client processes. We also use the local host address, 127.0.0.1, when we run the client and server processes on the same machine. When you test these programs in your environment, you can either specify the IP address of the machine that runs the server process or 12.0.0.1 if the client and server processes run on the same machine. Please note that you will not be able to run your client processes by using these IP addresses as command line arguments on your systems.

20.6.4.4 First Socket-Based Program

The `sockets.c` program shown in the following session creates a socket each for stream-oriented and datagram-oriented communication between processes on the Internet and displays the descriptors allocated in the PPFDT for these sockets. The compilation and execution of this program show that the socket descriptors 3 and 4 have been assigned to the two sockets. Note that descriptors 3 and 4 are the next two descriptors available after the standard descriptors.

```
% cat sockets.c
#include <sys/types.h>
#include <sys/socket.h>

int main(void)
{
        int s1, s2;

        if ((s1 = socket(PF_INET, SOCK_STREAM, 0)) == -1) {
            perror("SOCK_STREAM socket failed");
            exit(1);
        }
        if ((s2 = socket(PF_INET, SOCK_DGRAM, 0)) == -1) {
            perror("SOCK_DGRAM socket failed");
            exit(1);
        }
        printf("The socket descriptor for the stream socket is:
                %d\n", s1);
        printf("The socket descriptor for the datagram socket is:
                %d\n", s2);
        exit(0);
}
% gcc46 sockets.c -w -o sockets
% ./sockets
The socket descriptor for the stream socket is: 3
The socket descriptor for the datagram socket is: 4
%
```

On Solaris, you would generate the executable code using the `gcc  -w –lsocket sockets.c -o sockets` command.

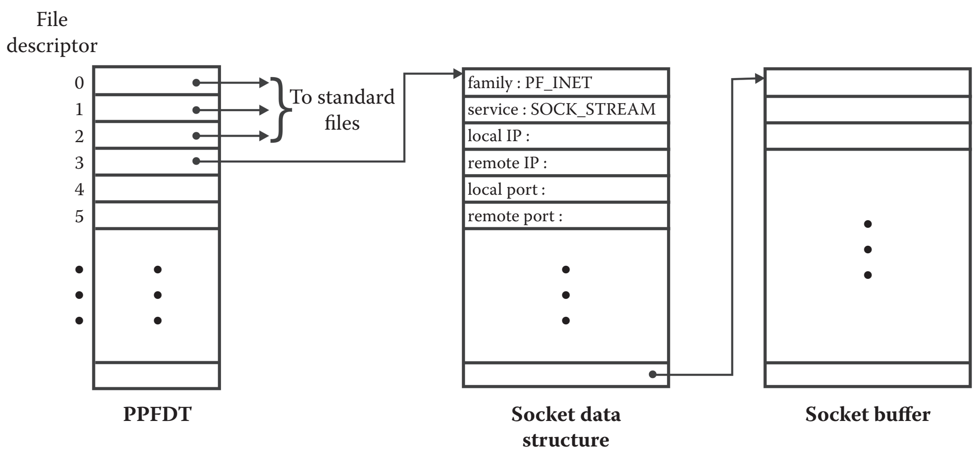


FIGURE 20.11 Partial description of socket descriptor, PPFDT, and socket data structure.

When an Internet domain socket is created, the UNIX kernel establishes a link between the socket descriptor for the newly created socket and the socket data structure allocated by the kernel when the socket is created. Figure 20.11 shows the high-level linkage between the socket descriptor and the associated socket data structure.

The socket data structure contains several pieces of information for the expected style of IPC, including domain, service type, local IP, remote IP, local port, and remote port. The UNIX kernel initializes the first two fields when a socket is created. The local IP and local port are stored in the data structure explicitly by the client or server process that creates it using the `bind()` system call. The <local IP, local port> pair forms the local address of the socket. When the local address has been stored in the socket data structure, we say that the socket is *half associated*. The remote IP and remote port fields are filled out when a client process calls the `connect()` system call. The <remote IP, remote port> pair is the address of the remote socket. When both local and remote addresses have been stored in the client- and server-side sockets, we say that *full association* has been established between the two processes. When full association has been established between two sockets, we say that they have been connected and a virtual connection has been established between them. For `SOCK _ STREAM` (i.e., TCP) sockets, this happens through the rendezvous of the `connect()` and `accept()` calls on the client and server sides, respectively, as discussed later in this chapter. Connection-oriented communication between two processes may take place only when full association exists between their sockets. We discuss socket addresses and their binding with sockets in more detail in the next two subsections.

20.6.4.5 Reading Data from a Socket

When a connection has been established between the server- and client-side sockets for the `SOCK _ STREAM` style of communication, they can communicate using the TCP protocol. This communication is based on a stream of bytes without any message boundaries. The sender process may send data using one or more `write()` calls. The receiver process may collect this data using a single or multiple `read()` calls. Suppose the server process needs to send 128 bytes of data to the client process and it does so by sending four 32-byte chunks using four `write()` system calls. The receiver side may receive all 128 bytes of data in one `read()` call or in multiple `read()` calls. It may read 4 bytes, 15 bytes, 78 bytes, 10 bytes, and 21 bytes in five successive `read()` calls. The number of bytes returned by a `read()` call depends on several factors, including the delays caused by the underlying network, the size of the datagrams in the underlying intranet, and the buffer space available associated with the sender- and receiver-side sockets. Thus, data from a `SOCK_STREAM` socket should be read in a loop until the desired amount of data has been received. The following code snippet shows one way of reading N bytes from a `SOCK_STREAM` socket and displaying the data on the screen.

```
char buf[N];
int n, nread, nremaining;

for (nread=0, n=0; nread < N; nread += n) {
    nremaining = N - nread;
    n = read(s, &buf[nread], nremaining);
    if (n == -1) {
        perror("read failed");
        exit(1);
    }
}
(void) write(1, buf, N);
```

The UDP protocol is used for communication between `SOCK_DGRAM` sockets. It is a "best-effort delivery service" protocol, under which messages are transmitted and received in terms of datagrams with fixed boundaries. Thus, a reader process tries to read the entire data sent by a writer process using a single `read()` system call. The process either reads the whole datagram or, in case of an error, does not read any data. The following code may be used to read from a `SOCK_DGRAM` socket.

```
n = read(s, buf, N);
if (n == -1) {
    perror("read failed");
    exit(1);
}
(void) write(1, buf, n);
```

20.6.5 Socket Address

The address (or name) of a socket is dependent on the socket domain. When a socket is created, it does not have any address. Until an address is assigned to a socket, it may not be referred to. For a UNIX domain (`PF_LOCAL` or `PF_UNIX`) socket, the address is a pathname in the file system. For an Internet domain (`PF_INET` or `PF_INET6`) socket, the address consists of two parts: the *IP address* of the host on which the socket is created and a *port number*. As discussed in Chapter 11, an IP address is used to uniquely identify a host on the Internet. IPv4 addresses are 32 bits long and IPv6 addresses are 128 bits long. On a host with a given IP address and running multiple servers, a port number allows the operating system kernel of the host to direct an incoming client request to a particular server on the host. Thus, the <IP address, port number> pair uniquely identifies the location of a service on the Internet, offered using different transport-layer protocols such as TCP and UDP.

A *port number* is a positive integer in the range 0 to 65535. The purpose of a port number is to distinguish different services offered on a host. The International Assigned Numbers Authority (IANA) assigns port numbers and service names. Service names are assigned on a first-come, first-served basis and port numbers are assigned according to a particular scheme, described in RFC6335 (see Table 20.26). Table 20.12 describes the general scheme used by IANA for the allocation of ports.

According to RFC6335, *system ports* and *user ports* are assigned by IANA using different processes. *Dynamic ports* are never assigned and may be used by any process randomly. System ports are used to offer well-known services and are, therefore, also known as *well-known ports*. Normally, the well-known ports are offered at the same port number regardless of the transport-level protocol used by the service (TCP, UDP, etc.). Note that only a superuser (i.e., **root**) can use ports < 1024. A few well-known services and their respective port numbers are listed in Table 20.13.

TABLE 20.12 Port Numbers and their Purpose

Port Number Range	Purpose
0–1023	System/well-known Ports
1024–49151	User/registered Ports
49152–65535	Dynamic/private Ports

TABLE 20.13 Some Well-Known Services and their Ports

Well-Known Service	Port
ECHO	7
DAYTIME	13
QOTD (Quote-of-the-Day)	17
FTP-DATA	20
FTP (File Transfer Protocol)	21
SSH (Secure Shell)	22
TELNET	23
SMTP (Simple Mail Transfer Protocol)	25
TIME	37
FINGER	79
HTTP, WWW	80
KERBEROS	88
POP3	110
SFTP	115

Typically, client processes and unknown/private servers use dynamic ports. These ports are also used to test server processes.

EXERCISE 20.6

Browse the IANA website. What is the total number of services that have been assigned port numbers?

EXERCISE 20.7

Browse through the **/usr/include/sys/socket.h** file. How many communication domains and service types have been defined?

20.6.6 Important Data Structures and Related Function Calls

UNIX provides many functions to network programmers to manipulate IP addresses. Most of these functions use data structures for storing socket names, that is, IP addresses, port numbers, address sizes, and other related information. A few are used to maintain information about a host on the Internet and an Internet service.

20.6.6.1 Important Data Structures for IP Addresses, Hosts, and Services

A brief summary of the most important data structures for network-based IPC and network programming is given in Table 20.3. These data structures deal with socket addresses, information about hosts on the Internet, and information about Internet services. We discuss the use of these data structures and their internal details in this section. The system calls, library calls, and macros that deal with them are discussed in the sections that follow.

`struct sockaddr`

This generic structure holds information about a socket's address. It is the basic template on which other address data structures used for storing addresses of sockets of different domains are based. Here is how the structure is defined in the **/usr/include/sys/socket.h** file.

```
struct sockaddr{
    unsigned short sa_family;
    char sa_data[14];
};
```

The `sa _ family` field contains the socket address family and the `sa _ data` contains the actual socket address. The value of `sa _ data` is interpreted based on the value of `sa _ family`. The address family used for UNIX domain sockets is `AF _ LOCAL` (or `AF _ UNIX`). For Internet domain sockets, it is `AF _ INET` or `AF _ INET6`. When `sa _ family` is `AF _ LOCAL`, the `sa _ data` field is supposed to contain a pathname as the socket's address. When `sa _ family` is `AF _ INET`, the `sa _ data` field contains both an IP address and a port number. The `sockaddr _ in` structure is specifically used for this purpose.

`struct sockaddr_un`

The address structure for the address of a UNIX domain socket is defined in **sys/un.h** as

```
struct sockaddr_un {
    unit8_t      sun_len;
    sa_family_t  sun_family;
    char         sun_path[104];
};
```

Here, `sun _ len` is the length of `sockaddr _ un` including the `NULL` byte, `sun _ family` is `AF _ LOCAL` (or `AF _ UNIX`), and `sun _ path[104]` is the null-terminated pathname in the file system structure that refers to the socket.

`struct sockaddr_in`

This structure may be used to hold the address information for an Internet domain socket. It is defined in the **/usr/include/sys/socket.h** file as follows:

```
struct sockaddr_in {
    uint8_t sin_len;
    sa_family_t sin_family;     /* short int */
    in_port_t int sin_port;     /* unsigned short */
    struct in_addr sin_addr;
    unsigned char sin_zero[8];
};
```

The `sin _ family` field specifies the address family; usually this is `AF _ INET`. The `sin _ port` and `sin _ addr` fields contain a 16-bit port number and a 32-bit IPv4 address in network byte order, respectively. The `sin _ zero` field is set to NULL (i.e., `'0'`) as it is not used. The `in _ addr` structure is defined as follows.

```
struct in_addr {
    unsigned long s_addr;
};
```

`struct hostent`

The `hostent` structure may be used to hold official information about a host on the Internet. It is defined in the **/usr/include/netdb.h** file as follows:

```
struct hostent
{
    char *h_name;
    char **h_aliases;
    int h_addrtype;
    int h_length;
    char **h_addr_list
#define h_addr h_addr_list[0] /* For backward compatibility */
};
```

The meaning of each field of the structure is given in Table 20.14.

`struct servent`

The `servent` structure may be used to hold official information about a host on the Internet. It is defined in the **/usr/include/netdb.h** file as follows:

```
struct servent
{
    char *s_name;
    char **s_aliases;
    int s_port;
    char *s_proto;
};
```

The meaning of each field of the structure is given in Table 20.15.

20.6.6.2 Library Functions to Manipulate IP Addresses

UNIX provides several library functions for manipulating IP addresses from ASCII, that is, strings in dotted decimal notation (DDN) to binary and vice versa. These functions are listed in Table 20.2. Here, we briefly discuss these calls and demonstrate their use with small code snippets.

TABLE 20.14 Meaning of Each Field of the hostent Structure

Field	**Purpose**
`h_name`	Official name of host
`h_aliases`	NULL-terminated array of other names of host
`h_addrtype`	Address type (family) of host, usually `AF_INET` (currently defined as `PF_INET`)
`h_length`	Length (in bytes) of address; 4 bytes for IPv4 and 16 bytes for IPv6
`h_addr_list`	NULL-terminated array of network addresses for host
`h_addr`	Used for backward compatibility; first address in `h_addr_list`

TABLE 20.15 Meaning of Each Field of the `servent` Structure

Field	Purpose
`h_name`	Official name of host
`h_aliases`	NULL-terminated array of other names of host
`h_addrtype`	Address type (family) of host, usually `AF_INET` (currently defined as `PF_INET`)
`h_length`	Length (in bytes) of address; 4 bytes for IPv4 and 16 bytes for IPv6
`h_addr_list`	NULL-terminated array of network addresses for host
`h_addr`	Used for backward compatibility; first address in `h_addr_list`

A brief description of the `inet _ aton()` call is given next. Note that it works for address domains `PF _ INET` and `PF _ INET6`.

```
#include <netinet/in.h>
#include <arpa/inet.h>
int inet_aton(const char *cp, struct in_addr *pin);
```
Success: `1`
Failure: `0` if address string is invalid

This function call converts the IP address in DDN specified as string `cp` to the IP address in network byte order and stores it in the structure specified as `pin`. The following code snippet shows sample use of this function.

```
#include <netinet/in.h>
#include <arpa/inet.h>
...
int n;
struct in_addr address;
...
memset(&address, '\0', sizeof(address));
n = inet_aton("39.59.169.110", &address);
if (n == 0) {
   /* Error handling code */
}
...
```

The `inet _ addr()` call converts and returns the specified string for an IP address in DDN to a 32-bit unsigned binary value in network byte order. Here is a brief description of the call.

```
#include <sys/types.h>
#include <arpa/inet.h>
in_addr_t inet_addr(const char *cp);
```
Success: IP address in Network Byte Order
Failure: `INADDR_NONE`

The return type of the function, `in _ addr _ t`, is defined as a 32-bit unsigned integer in the **/usr/include/sys/types.h** file. In the following code segment, we show a sample use of this call.

```
#include <sys/types.h>
#include <arpa/inet.h>
...
struct sockaddr_in address;
...
memset(&address, '\0', sizeof(address));
if ((address.sin_addr.s_addr = inet_addr("39.59.169.110")) ==
     INADDR_NONE) {
    /* Error handling code */
}
...
```

The call returns `INADDR _ NONE` on failure. This symbolic constant is defined in **/usr/include/netinet/in.h** as all 1s (i.e., `0xffffffff`). This means that this return value is `-1`.

The `inet _ ntoa()` call converts and returns an IP address in network byte order binary form to the string of corresponding IP address in DDN. Here is a brief description of the call.

```
#include <sys/types.h>
#include <arpa/inet.h>
char * inet_ntoa(struct in_addr in);
```
Success: IP address string in DDN
Failure: `NULL`

The following piece of code shows a sample use of the call.

```
#include <arpa/inet.h>
#include <arpa/inet.h>
...
char *ip_ddn;
struct in_addr address;
...
ip_ddn = inet_ntoa(address);
printf("IP Address is: %s\n",ip_ddn);
...
```

20.6.6.3 New Library Functions to Manipulate IP Addresses

The `inet _ addr()` function received criticism and new programs used `inet _ aton()` instead. The new versions of `inet _ ntoa()` and `inet _ aton()` that work with both IPv4 and IPv6 addresses are `inet _ ntop()` and `inet _ pton()`, respectively. Note that "n" stands for numeric and "p" for presentation. You should use the new calls in your code even if your system does not support IPv6.

Here are brief descriptions of the `inet _ ntop()` and `inet _ pton()` functions.

```
#include <sys/types.h>
#include <arpa/inet.h>
const char *inet_ntop(int af, const void * restrict src, char *
 restrict dst, socklen_t size);
```

Success: IP address string in DDN
Failure: NULL on system error; `-1` the specified address family not supported

```
#include <sys/types.h>
#include <arpa/inet.h>
int inet_pton(int af, const char * restrict src, void * restrict dst);
```

Success: `1`
Failure: `0` if input is not in a valid presentation format and `-1` on error

The first argument in both functions, `af`, stands for address family. Currently, only `AF _ INET` and `AF _ INET6` are supported. These functions return `-1` if the specified address family is not supported and `errno` variable is set to `EAFNOSUPPORT`. The size argument in `inet _ atop()` is the size in bytes of the destination. It is used to prevent overflow of caller function's buffer. If the `size` argument is too small to store the address in the resultant presentation, the function returns `NULL` and `errno` is set to `ENOSPC`. To prevent this problem, the following constants, defined in **/usr/include/arpa/inet.h**, should be used.

```
#define INET_ADDRSTRLEN      16    /* for IPv4 dotted-decimal */
#define INET6_ADDRSTRLEN     46    /* for IPv6 hex string */
```

Now, we show a few examples of the old calls and their equivalents using the new calls. The first line shows the old call and the second (and third) shows its equivalent of the new call.

```
address.sin_addr.s_addr = inet_addr("39.59.169.110");
inet_pton(AF_INET, "39.59.169.110", address.sin_addr);
inet_aton("39.59.169.110", &address);
inet_pton(AF_INET, "39.59.169.110", address.sin_addr);
ip_ddn = inet_ntoa(address);
char dest[INET_ADDRSTRLEN];
ip_ddn = inet_ntop(AF_INET, &address, dest, sizeof(dest));
```

EXERCISE 20.7

Browse through the header files that contain the data structures that we have discussed in this section.

EXERCISE 20.8

Write small programs to test the use of the various library calls and macros that we have discussed in this section. Show compilation and execution of your programs along with their outputs.

20.6.7 Binding an Address to a Socket

When a socket is created, it belongs to a particular protocol domain but does not have any protocol address assigned to it. The protocol address of a socket is also known its *address* or *name*. If no process would refer to a socket, then it is not necessary for such a socket to have an address. For example, if a socket were created in a client process, then, most likely, no other process would refer to it. Thus, binding an address to a client-side socket is not necessary. If other processes would need to refer to a socket, it is necessary that it have an address. A socket created by a server process requires that an address be bound (i.e., assigned) to it. This address is advertised for the service that the server process offers so that a client process could contact the server process using the address of the server-side socket. Figure 20.12 shows a server process with three sockets having addresses and a client process with a socket without an address assigned to it. The sockets in the server process are of the types PF _ LOCAL, PF _ INET with SOCK _ STREAM type of communication, and PF _ INET with SOCK _ DGRAM type of communication. The client socket does not have an address.

You can use the bind() system call to assign an address to a socket. Here is a brief description of the bind() system call.

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int s, const struct sockaddr *addr, socklen_t addrlen);
```
Success: 0
Failure: -1 and kernel variable errno set to indicate the type of error

Here, s is a socket descriptor, addr is the address to be bound to s, and addrlen is the length of addr. For maximum portability, the addr field must be initialized to zero.

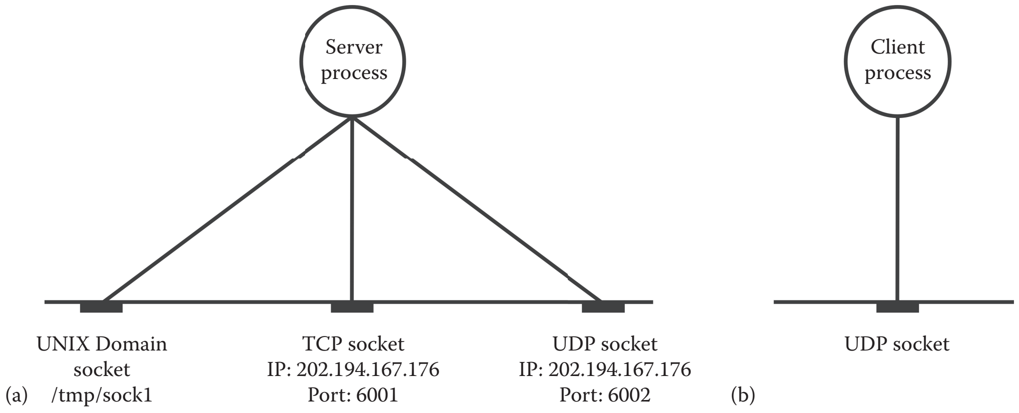

Server process
Client process
UNIX Domain socket /tmp/sock1
(a)
TCP socket IP: 202.194.167.176 Port: 6001
UDP socket IP: 202.194.167.176 Port: 6002
(b)
UDP socket

FIGURE 20.12 Sockets with and without addresses bound to them: (a) server process with three sockets with addresses; a PF _ LOCAL socket, a PF _ INET socket of SOCK _ STREAM type, a PF _ INET socket of SOCK _ DGRAM type; and (b) client process with PF _ INET socket of SOCK _ DGRAM type without address.

TABLE 20.16 Common Reasons for the `bind()` System Call to Fail

Reason for Failure	Value of `errno`
Sufficient kernel resources not available to complete the request	EAGAIN
The `s` argument is not a valid descriptor	EBADF
The `s` argument represents a socket that has been shut down (or closed) or is already bound to an address	EINVAL
The `s` argument does not represent a socket	ENOTSOCK
The `addr` argument represents an address that is already in use	EADDRINUSE
The `addr` argument is not within the process address space	EFAULT

The `bind()` call may fail for various reasons. Some of the commonly occurring reasons are listed in Table 20.16.

In case of the UNIX domain socket, the `bind()` system call would fail for the reasons that the `open()`, `creat()`, `mkfifo()`, and `mkfifoat()` system calls would fail, as shown in Tables 18.4 and 20.7. When a UNIX domain socket is no longer required, it must be removed from the system using the `unlink()` system call. The address of a socket cannot be read/written by anyone other than the processes with which the socket is associated.

The following code snippet shows how you can use the `bind()` system call to bind address to an Internet domain socket.

```
#define PORT  6001
...
struct sockaddr_in saddr, caddr;
...
/* Initialize socket structure */
memset(&saddr, 0, sizeof(saddr));
saddr.sin_family =  AF_INET;
saddr.sin_addr.s_addr =  INADDR_ANY;
saddr.sin_port =  htons(PORT);

/* Bind address to socket */
if (bind(s, (struct sockaddr *)&saddr, sizeof(saddr)) == -1) {
    perror("bind failed");
    exit(1);
}
```

Note that the address structure variable must be properly initialized before using it in the `bind()` call. The symbolic constant `INADDR _ ANY`, defined in **/usr/include/netinet/in.h**, is a wildcard IP address that matches any of the IP addresses of the host on which the server process runs. Thus, in case of *multihomed hosts*, that is, hosts that are connected to multiple networks and, hence, have multiple IP addresses, the use of the wildcard address `INADDR _ ANY` makes it possible for the server to accept connection requests from clients arriving at any of these IP addresses. The `htons(PORT)` function is used to convert the PORT value from the host network byte order to network byte order before assigning the port value to the `sin _ port` field of the `saddr` address variable. Only a superuser (i.e., **root**) can bind to ports < 1024.

EXERCISE 20.9

Write a program that creates a UNIX domain socket, assigns a name to it, and displays the socket descriptor and its address using the UNIX address variable.

20.6.8 Enabling a Server-Side Socket to Listen for Connection Requests from Clients

Once a server-side stream-oriented socket has been assigned an address, it needs to be put in passive (listening) mode before client processes may connect to and communicate with it. You can put a socket in passive mode using the `listen()` system call. Once the `listen()` call has been called on a socket, it starts to listen for incoming connection requests from client processes. Such a socket is known as a *passive socket*. Once a socket has been placed in passive mode, a client request for connection may be accepted by it.

Here is a brief description of the `listen()` system call.

```
#include <sys/types.h>
#include <sys/socket.h>
int listen(int s, int backlog);
```

Success: 0
Failure: -1 and kernel variable `errno` set to indicate the type of error

Here, `s` is a socket descriptor and `backlog` specifies the maximum length of the queue where client connection requests may wait. According to the manual page for the `listen()` call, the real maximum queue length is 1.5 times what the programmer specifies as `backlog`. The maximum length of this queue is defined in the **/usr/include/sys/socket.h** file as `SOMAXCONN`. This value of `SOMAXCONN` is 128 on BSD and Solaris. Several well-known servers use the maximum queue length with their passive sockets. You can use the `netstat -aL` command to determine the queue lengths associated with all the servers running on your system.

The `listen()` call may fail for the reasons listed in Table 20.17.

TABLE 20.17 Reasons for `listen()` to Fail

Reason for Failure	Value of `errno`
`s` is not a valid descriptor	`EBADF`
`s` is not a socket descriptor	`ENOTSOCK`
`s` is already connected or is in the process of being connected	`EINVAL`
`s` is of a type that does not support the `listen()` operation	`EOPNOTSUPP`

The following code snippet shows a typical use of the `listen()` system call. Note that the queue length associated with the passive socket is 5.

```
#define QLEN  5
...
/* Put socket in passive mode */
if (listen(s, QLEN) == -1) {
    perror("listen failed");
    exit(1);
}
...
```

20.6.9 Sending a Connection Request to Server Process

In the connection-oriented style of communication, the server- and client-side sockets are first connected, and only then does communication start between the two processes through their respective sockets—both processes do I/O with their own sockets. Once the server-side stream-oriented socket has been assigned an address and put in passive mode, it is ready to receive and accept client requests to establish connection. A client process can send a connection request using the `connect()` system call.

Here is a brief description of the `connect()` system call.

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int s, const struct sockaddr *name, socklen_t namelen);
```
Success: 0
Failure: -1 and kernel variable `errno` set to indicate the type of error

For maximum portability, the `addr` field must be initialized to zero.

When used with the `SOCK _ STREAM` type of socket, the `connect()` call performs the three-way TCP handshake to connect the client- and server-side sockets. After the call has completed its execution, the underlying data structure associated with each socket has the address of the other side and a full association has been established between the two sockets. This also means that a *virtual connection* has been established between the client and server processes. This only happens through the rendezvous of the `connect()` and `accept()` system calls. We discuss the `accept()` system call in the next section. Once a connection has been established between the client- and server-side sockets, the client and server processes can communicate with each other using the `read()` and `write()` system calls.

When used with the `SOCK _ DGRAM` type of socket, the `connect()` call simply stores the address of the remote socket in the local socket's data structure. No handshake takes place between the local and remote sockets. This means that after a UDP client has made a `connect()` call on a socket, it may communicate with the other side using `read()` and `write()` instead of using `recvfrom()` and `sendto()`.

TABLE 20.18 Some Reasons for connect() to Fail

Reason for Failure	Value of errno
s is not a valid descriptor	EBADF
s is not a socket descriptor	ENOTSOCK
Address specified in name is not available on the computer	EADDRNOTAVAIL
Address specified in name cannot be used with this socket	EAFNOSUPPORT
s is already connected	EISCONN
Host is not connected to the network	ENETUNREACH
Remote host is not reachable from this host	EHOSTUNREACH
Address specified in name is already in use	EADDRINUSE
A signal interrupted the connection establishment	EINTR
A previous connection request has not yet been completed	EALREADY
A broadcast address is specified using INADDR_BROADCAST or INADDR_NONE for a socket that does not support broadcast functionality	EACCES

Normally, connect()is called only once to successfully establish full association between two SOCK _ STREAM sockets. However, connect() may be called multiple times to change associations between two SOCK _ DGRAM sockets. An association of a SOCK _ DGRAM socket may be dissolved by using connect() on it with an invalid address, such as a null address.

The connect() call may fail for several reasons. Table 20.18 shows some of the reasons that the connect() call may fail for non-UNIX domain—usually PF _ INET or PF _ INET6—sockets.

Failure of the connect() call for a UNIX domain socket has mostly to do with invalid pathname issues, as discussed for system calls such as open() and creat(). Two additional reasons for failure of the connect() call for a UNIX domain socket are: (a) the socket s does not exist, and (b) write access for s is denied.

The following piece of code shows how you can use the connect() system call to send a connection request to a connection-oriented server process to establish a connection with it. We assume that, as is the case with all production clients, the name of the host or its IP address in DDN and the port number on which the server process runs are passed as the first and second command line arguments to the client program.

```
...
struct sockaddr_in saddr;
struct hostent *server;
...
if ((server = gethostbyname(argv[1])) == NULL) {
     printf("No such host\n");
     exit(0);
}

memset(&saddr, 0, sizeof(saddr));
saddr.sin_family = AF_INET;
saddr.sin_port = htons(atoi(argv[2]));
```

```
if (server = gethostbyname(argv[1]))
     memcpy(&saddr.sin_addr.s_addr, server->h_addr,
              server->h_length);
else
     if ((saddr.sin_addr.s_addr = inet_addr(argv[1])) ==
          INADDR_NONE) {
     printf("No such host\n");
     exit(1);
     }

/* Send connection request to the server process */
if ((connect(s, &saddr, sizeof(saddr))) == -1) {
     perror("connect failed");
     exit(1);
}
...
```

Note that gethostbyname() returns a pointer to the hostent structure, which contains relevant information about the host, including its address and address length, as discussed in Section 20.6.6. We copy the address of the host to the relevant field of the address variable for the server-side socket, saddr, using the bcopy() or the newer memcpy() function. The atoi() function received criticism and the newer recommended replacement is strtol(). We use the strtol() function in the client–server software that we design and develop in this chapter. You can also replace

```
saddr.sin_addr.s_addr = inet_addr(argv[1]);
```

with the following call:

```
inet_pton(AF_INET, argv[1], address.sin_addr);
```

20.6.10 Accepting a Client Request for Connection

Whether it is used on an *iterative server* or a *concurrent server*, the passive socket may only be used to wait for client requests for connections and not for client–server communication. This is so because, after accepting a client's request for connection and before starting communication with the client, the server process needs to go back and listen for more incoming connection requests from clients. Thus, a client-side socket does not communicate with the passive socket on the server side. Instead, a new socket is created on the server side that is connected with the client-side socket for communication between the two processes. Because it is used to interact with the client process, the newly created socket is also known as the *active socket*. As stated earlier, this new socket on the server side is created and connected to the client-side socket through the rendezvous of the accept() call on the server side and the connect() call on the client side.

The main server socket—the passive socket—remains allocated throughout the life of a server process. However, the sockets created by accept() have the life span of a connection between a client and the server process. For this reason, they are also known as *ephemeral sockets*.

A variant of the `connect()` call is `accept4()`. Here is a brief description of the `accept()` and `accept4()` system calls.

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int s, struct sockaddr * restrict addr, socklen_t * restrict
           addrlen);
int accept4(int s, struct sockaddr * restrict addr, socklen_t *
            restrict addrlen,int flags);
```

Success: Socket descriptor for the accepted socket
Failure: –1 and kernel variable `errno` set to indicate the type of error

Here, `s` is the server-side passive socket, `addr` is the address variable, and `addrlen` is the length of the address variable in bytes. The call returns the descriptor for the newly created socket, which inherits the properties of `s` for nonblocking and asynchronous I/O (`O _ NONBLOCK` and `O _ ASYNC`), and settings of the I/O and urgent signals (`SIGIO` and `SIGURG`). For the `accept4()` call, the nonblocking I/O property is specified by using the `SOCK _ NONBLOCK` flag in the `flags` argument. The "close-on-exec" property of the newly created socket is set using the `SOCK _ CLOEXEC` flag in the `flags` argument. The signal settings and asynchronous I/O properties of the newly created socket are cleared. For maximum portability, the `addr` field must be initialized to zero.

The `accept()` and `accept4()` calls may fail for several reasons. Table 20.19 lists some of the reasons for these calls to fail.

The following code snippet shows how you can use the `accept()` system call to wait, listen for, and accept connection requests from an Internet client through its socket.

```
...
int caddrlen;
struct sockaddr_in caddr;
...
```

TABLE 20.19 Some Reasons for `accept()` and `accept4()` to Fail

Reason for Failure	Value of `errno`
`s` is not a valid descriptor	`EBADF`
A signal interrupted the accept operation	`EINTR`
The PPFDT or the system-wide file table is full	`EMFILE`
`s` is not a socket descriptor	`ENOTSOCK`
`s` is not a passive socket, i.e., the `listen()` system call has not been executed on `s`. In the case of the `accept4()` call, the `flags` argument is invalid	`EINVAL`
`addr` is not the writable part of the address space of the caller process	`EFAULT`
`s` is nonblocking with no connection requests waiting to be accepted	`EWOULDBLOCK`

```
/* Obtain length of client's address variable */
caddrlen = sizeof(caddr);
/* Block, listen for, and accept a connection request from a
   client */
if ((sock = accept(s, (struct sockaddr *)&caddr, &caddrlen))
     == -1) {
    perror("accept failed");
    exit(1);
}
...
```

Figure 20.13 shows the pictorial view of the complete client–server setup for the `SOCK _ STREAM` type of communication using a single server process. The figure clearly shows the sequence of steps that the client and server processes have to take for the establishment of a virtual connection between the client- and server-side sockets. We discuss similar setups for multiprocess servers later in the chapter.

20.6.11 Closing a Socket

When a socket is no longer needed, it should be closed so that its descriptor and associated kernel resources may be reused. The `close()` or `shutdown()` system calls may be used to close a socket. The `close()` system call closes the socket completely, whereas the

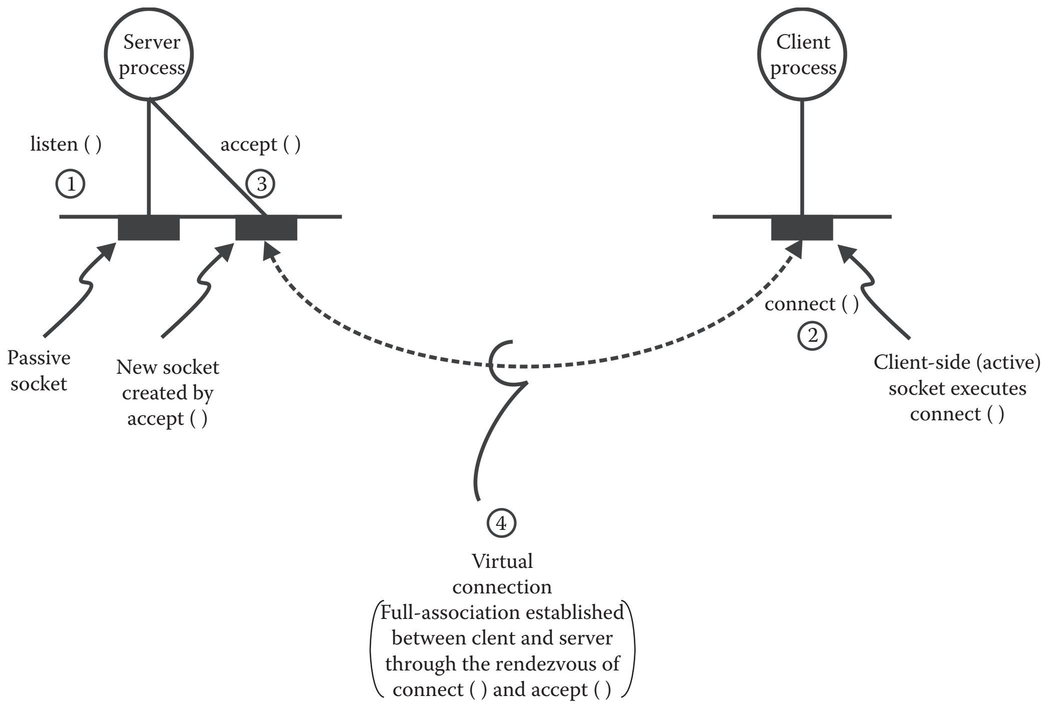

Server process
Client process
listen ( )
1
accept ( )
3
connect ( )
2
Passive socket
New socket created by accept ( )
Client-side (active) socket executes connect ( )
4
Virtual connection
(Full-association established between clent and server through the rendezvous of connect ( ) and accept ( ))

FIGURE 20.13 View of the passive socket and virtual connection between client and server sides.

TABLE 20.20 Possible Values of the how Parameter and Its Effect on Socket s

Value of how	Effect on Socket s
SHUT_RD	No input (i.e., read) operation may be performed on socket s
SHUT_WR	No output (i.e., write) operation maybe performed socket s
SHUT_RDWR	No I/O may be performed on socket s

shutdown() call may be used to close a socket for input (read) only, output (write) only, or both input and output.

Here is a brief description of the shutdown() system call.

```
#include <sys/types.h>
#include <sys/socket.h>
int shutdown(int s, int how);
```

Success: 0
Failure: -1 and kernel variable errno set to indicate the type of error

Table 20.20 shows the different values for the how parameter and its effect on the socket s.

The shutdown() call is used when a process has completed either input or output, but not both. For example, a client process can close its socket for output after sending its last request to the server process. The server process may close its socket for both input and output after it has sent its response to the last client request it has received. Finally, the client process closes the socket for input after receiving the response to its last request.

20.6.12 Putting it All Together: A Simple Connection-Oriented Client–Server Software

We now put together the code snippets shown for the various system calls and build simple connection-oriented client–server software. This service is similar to the well-known ECHO service, except that the server process does not run at the well-known port 7, and terminates after serving one client request.

20.6.12.1 Design of a Server Process

In the design of our code for the client and server processes, we write some generic functions that may be used in the TCP and UDP client and server processes. The server processes we discuss in this section run with two command line arguments, the transport-level protocol for which the service is offered (tcp, udp, etc.) and the protocol port at which the service is offered. Thus, the syntax for the execution of a server process is shown next.

```
server-name transport-protocol protocol-port
```

If transport-protocol or protocol-port is invalid, the program displays an error message and terminates.

The first function that we will design is `CreatePassiveSock()`. It takes three arguments: a transport protocol, a protocol port, and the queue length for the passive socket. The protocol and port are passed to the server process as command line arguments. The server supports only TCP and UDP as transport-level protocols. It creates a socket for the type of communication for the given protocol, binds a name to the socket, and puts the socket in passive mode if it is a TCP socket. Finally, it returns the descriptor for the socket as its return value.

As discussed earlier in Section 20.6.7, the wild card IP address `INADDR _ ANY` makes it possible for the server to accept connection requests from clients at any of the IP addresses of a multihomed host. Note that we use the `strtol()` library call to convert a port number string to an integer and make sure to exit if a string of nondigits is passed as the port number at the command line. The criticized `atoi()` function would not work properly because it converts some character strings of nondigits into valid port numbers.

The second function, `EchoServerTCP()`, is specific to the service to be offered by the server process. It takes an active socket created by the `accept()` system call as an argument, reads data from the client process, and writes it back to the client process. Normally, we would read data from a TCP socket in a loop. However, to keep things simple, we read client data using a single `read()` call. If you want to make it a production server, you must read and write data in a loop as discussed in Section 20.6.4.

Here is the code for the server process saved in the **echo_server_TCP.c** file.

```
% cat echo_server_TCP.c
/*
Usage: Server-name Protocol Port
Here, Protocol is the transport level protocol
and Port is the procotol port number where the
service is to be offered.
*/

#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define BUFF      256
#define QLEN      16

int main(int argc, char *argv[])
{
    int psock, asock, caddrlen;
    struct sockaddr_in caddr;

    /* Exit if program is not run with two command line
       arguments. */
    if (argc != 3) {
        printf("Usage: server protocol port\n");
        exit(1);
    }

    /* Create a TCP socket, bind name to it, and put it in passive
       mode */
    psock = CreatePassiveSock(argv[1], argv[2], QLEN);

    caddrlen = sizeof(caddr);
    while (1) {
        /* Accept actual connection from the client */
        if ((asock = accept(psock, (struct sockaddr *)&caddr,
                            &caddrlen)) == -1) {
            perror("accept failed");
            exit(1);
        }

        /* TCP echo service code */
        EchoServerTCP(asock);

        /* Close active socket */
        close(asock);
    } /* while */
}

/* Create a TCP or UDP socket, bind a name to it, and put it in */
/* passive mode if it is a TCP socket.                           */
/* 'protocol' is transport layer protocol ("tcp", "udp", etc).  */
/* 'portptr' is pointer to port number as a character string.   */
/* 'qlen' is the queue length associated with the passive socket. */
int CreatePassiveSock(char *protocol, char *portstr, int qlen)
{
    int s, port, type, saddrlen;
    char *endptr;
    struct sockaddr_in saddr;

    /* Convert portstr to port number as integer. Display  */
    /* error message and exit if portstr is not a number.  */
    port = (int) strtol(portstr, &endptr, 10);
    if (*endptr) {
        printf("\nPlease specify a positive integer for port.\n");
        exit(1);
    }

    /* Initialize socket structure */
    saddrlen = sizeof(saddr);
    memset(&saddr, 0, saddrlen);
    saddr.sin_family = AF_INET;
    saddr.sin_addr.s_addr = INADDR_ANY;
    saddr.sin_port = htons(port);

    if (strcmp("tcp", protocol) == 0)
        type = SOCK_STREAM;
    else if (strcmp("udp", protocol) == 0)
        type = SOCK_DGRAM;
    else {
        printf("Unsupported protocol\n");
        exit(1);
    }

    /* Create a TCP or UDP socket for IPv4 */
    if ((s = socket(PF_INET, type, 0)) == -1) {
        perror("socket call failed");
        exit(1);
    }

    /* Bind address to socket */
    if (bind(s, (struct sockaddr *)&saddr, saddrlen) == -1) {
        perror("bind failed");
        exit(1);
    }

    /* If it is a TCP socket, put it in passive mode, */
    /* i.e., ready to listen for incoming connection  */
    if (type == SOCK_STREAM) {
        if (listen(s, qlen) == -1) {
            perror("listen failed");
            exit(1);
        }
    }

    /* Return the TCP passive or UDP socket with a    */
    /* name bound to it.                              */
    return s;
}

/* Provide echo service to a client. 'sock' is the     */
/* active socket connected to the client-side socket.  */
void EchoServerTCP(int sock)
{
    int nr, nw;
    char buf[BUFF];

    /* Communicate with client: read and write back */
    memset(buf, 0, BUFF);
    if ((nr = read(sock, buf, BUFF-1)) == -1) {
        perror ("socket read error");
        exit(1);
    }

    /* Write back (echo) the same data to client */
    if ((nw =  write(sock, buf, nr)) == -1) {
        perror("socket write error");
        exit(1);
    }
}
%
```

First, the program makes sure that it has been run with the correct number of arguments. Second, it verifies that the second argument, `port _ number`, is a number. It then calls the `CreatePassiveSock()` function to create a passive socket in the case of the TCP protocol. Finally, it starts an infinite loop and blocks on the `accept()` call. As soon a client request arrives and has been accepted, an active socket is created with its descriptor in `asock`. The program then calls `EchoServerTCP()` to service the client request, passing it the active socket's descriptor as an argument. This function serves the client and returns. On return from this function, the program closes the active socket and blocks on `accept()` again, waiting for the next connection request.

Here is the compilation and a sample run of the client–server model. On a Solaris machine, you would use `gcc -w -lsocket echo _ server _ TCP.c -o tcpechos` to create the executable file. Note that in our sample run, we offer the TCP ECHO service at port 6001. The output of the `netstat –a | head` command shows our server running on port 6001. The –a option is used to also show the states of sockets associated with all server processes. Since we have not yet written the code for the client process corresponding to this server, we test it with a **telnet** client. The **telnet** client sends the text entered by the user from the keyboard (shown in boldface) to the server process, the server process receives it, sends it back to the **telnet** client, deallocates the active socket, and goes back to wait for another client's connection request. The **telnet** client receives the text from the server process, displays it on the screen, and terminates. You can use the loopback address (127.0.0.1) with the `telnet` command instead of using the IP address of the host (202.147.169.196), because the client and server processes run on the same host. The last `ps` command is used to show that after serving a client, the server continues to run, as expected. The `kill` command is used to terminate the server process.

```
% gcc46 echo_server_TCP.c -w -o tcpechos
% ./tcpechos tcp 6001 &
[1] 33690
% ps
  PID TT  STAT    TIME COMMAND
 8437  1  Ss   0:01.11 -csh (csh)
33690  1  S    0:00.00 ./tcpechos tcp 6001
33691  1  R+   0:00.01 ps
% netstat -a | head
```

```
Active Internet connections (including servers)
Proto Recv-Q Send-Q Local Address    Foreign Address      (state)
tcp4     0          0 *.6001           *.*                  LISTEN
tcp4     0          0 202.147.169.196.ssh  39.59.72.116.60434
                                                         ESTABLISHED
tcp4     0          0 *.ssh            *.*                  LISTEN
[output truncated]
% telnet 202.147.169.196 6001
Trying 202.147.169.196...
Connected to 202.147.169.196.
Escape character is '^]'.
Hello, world!
Hello, world!
Connection closed by foreign host.
% ps
  PID TT  STAT    TIME COMMAND
 8437  1  Ss   0:01.13 -csh (csh)
33690  1  S    0:00.00 ./tcpechos tcp 6001
33725  1  R+   0:00.01 ps
% kill -9 33690
%
```

EXERCISE 20.10

Repeat this session on your UNIX system to verify that it works as expected.

20.6.12.2 Design of a Client Process

The client processes that we discuss in this chapter run with three command line arguments, the IP address in DDN or domain name of the server process, the protocol port at which the service is offered, and the transport-level protocol for which the service is offered (tcp, udp, etc.). Thus, the syntax for the execution of a client process is shown next.

```
client-name ip-address (or domain name) protocol-port
                                              transport-protocol
```

If any of the command line arguments is invalid, the program displays an error message and terminates.

The first function that we design for the client software is CreateConnectedSock(), which takes the three arguments that you pass to the client process as command line arguments. It converts the port number from a string to an integer using the strtol() library call and initializes the address variable for the server process. It then creates a TCP or UDP socket, depending on the value of the transport-level protocol. Next, it establishes a connection with the server-side socket using the connect() system call and returns the socket descriptor. As has been discussed earlier, in the case of the UDP client–server software, no connection is established between the client- and server-side processes. However, the server's address is stored in the client-side socket's data structure. This allows the client to communicate with the server process using the `read()` and `write()` system calls, instead of the `sendto()` and `recvfrom()` system calls.

The second function, `EchoClientTCP()`, handles the client side of the ECHO service. It takes a connected socket descriptor at the client side as an argument, prompts the user for input from the keyboard, writes the user input to the server process, reads the server's response (which is the client-side data sent back), displays it on the screen, and returns. As stated earlier, normally, we would read and write data from and to a TCP socket in a loop. However, to keep things simple for this rather trivial service, we do I/O with the socket using single `read()` and `write()` calls. If you want to make it part of a production client–server software, you must read and write data in loops as discussed in Section 20.6.4. Here is the code for the function.

Here is the code for the server process saved in the **echo_client_TCP.c** file. After making sure that the client program is run with the requisite number of command line arguments, it calls the `CreateConnectedSock()` function, which returns the descriptor for the appropriate socket connected to the server-side socket. It then calls the `EchoClientTCP()` function to perform the client-side functionality of the ECHO service. At the end, it deallocates the connected socket and returns.

```
% cat echo_client_TCP.c
#include <netdb.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define BUFF    256

int main(int argc, char *argv[])
{
    int csock; /* Descriptor for connected socket */

    if (argc != 4) {
        printf("Usage: %s hostname port protocol\n", argv[0]);
        exit(0);
    }

    csock = CreateConnectedSock(argv[1], argv[2], argv[3]);

    /* Perform client-side echo using csock, the connected socket */
    EchoClientTCP(csock);

    /* Deallocate socket and return */
    close(csock);
    return 0;
}
```

```
/* Create a TCP or UDP socket, based on the third command line  */
/* argument. Connect the socket to the server whose name (IP    */
/* address and port number) is passed as command line arguments. */
/* Return the descriptor of the connected socket. In case of the */
/* UDP socket is not connected but the address of the remote    */
/* socket is stored in the local socket's data structure. This  */
/* allows us to communicate with the server using the read()    */
/* and write() system calls, instead of recvfrom() and sendto(). */
int CreateConnectedSock(char *ip, char *portstr, char *protocol)
{
    int s, port, type, saddrlen;
    char *endptr;
    struct sockaddr_in saddr;
    struct hostent *server;

    /* Convert portstr to port number as integer. Display  */
    /* error message and exit if portstr is not a number.  */
    port = (int) strtol(portstr, &endptr, 10);
    if (*endptr) {
        printf("\nPlease specify a positive integer for port.\n");
        exit(1);
    }

    saddrlen = sizeof(saddr);
    memset(&saddr, 0, saddrlen);
    saddr.sin_family = AF_INET;
    saddr.sin_port = htons(port);
    if (server = gethostbyname(ip))
        memcpy(&saddr.sin_addr.s_addr, server->h_addr,
               server->h_length);
    else
        if ( !(inet_pton(AF_INET, ip, saddr.sin_addr)) ) {
            printf("No such host\n");
            exit(1);
        }

    if (strcmp("tcp", protocol) == 0)
       type = SOCK_STREAM;
    else if (strcmp("udp", protocol) == 0)
       type = SOCK_DGRAM;
    else {
       printf("Unsupported protocol\n");
       exit(1);
    }

    /* Create a TCP or UDP socket for IPv4 */
    if ((s = socket(PF_INET, type, 0)) == -1) {
        perror("socket call failed");
        exit(1);
    }

    /* Send connection request to the server process */
    if (connect(s, &saddr, saddrlen) == -1) {
        perror("connect failed");
        exit(1);
    }
    return s;
}

/* Client side of the echo service. 'sock' is the  */
/* socket connected to the server-side socket.     */
void EchoClientTCP(int sock)
{
    int n;
    char buf[BUFF];

    /* Get input from the user */
    memset(buf, 0, BUFF);
    printf("Enter text for server : ");
    fgets(buf, BUFF-1, stdin);

    /* Send message to server */
    if ((n =  write(sock, buf, strlen(buf))) == -1) {
        perror("socket write error");
        exit(1);
    }

    /* Rread server's response */
    memset(buf, 0, BUFF);
    if ((n = read(sock, buf, BUFF-1)) == -1) {
        perror ("socket read failed");
        exit(1);
    }

    /* Display server's response */
    printf("Text from server : %s", buf);
}
%
```

In the following session, we show the compilation of the client software and the running of the executable codes for the server and client processes. The client–server software runs as expected. Again, you can use the loopback address (127.0.0.1) with the client command instead of using the IP address of the host (202.147.169.196), because the client and server processes run on the same host.

```
% gcc46 echo_client_TCP.c -w -o tcpechoc
% ./tcpechos tcp 6001 &
```

```
[1] 27230
% ps
  PID TT  STAT    TIME COMMAND
 8437  1  Ss   0:00.74 -csh (csh)
27230  1  S    0:00.00 ./tcpechos tcp 6001
27237  1  R+   0:00.01 ps
% ./tcpechoc 202.147.169.196 6001 tcp
Enter text for server : Hello, world!
Text from server : Hello, world!
% kill -9 27230
%
```

Because the client program uses library calls gethostbyname() and inet _ pton(), the compilation of the program on Solaris requires linking with the -lsocket and -lnsl switches, as shown next. We also show a few sample runs of the client program.

```
$ gcc -lsocket -lnsl echo_client_TCP.c -w -o tcpechoc
$ ./tcpechoc 127.0.0.1 6001 tcp
Enter text for server : Hello, world!
Text from server : Hello, world!
$ ./tcpechoc 202.147.169.197 6001 tcp
Enter text for server : Hello, world!
Text from server : Hello, world!
$
```

EXERCISE 20.11

Repeat this session on your UNIX system to verify that it works as expected.

EXERCISE 20.12

Run the client and server processes in different windows on the same machine and on different machines on the Internet.

EXERCISE 20.13

Design and code the client–server model for the ECHO service using UNIX domain sockets. Show the compilation and execution of the model on your machine.

20.7 TYPES OF SOCKET-BASED SERVERS

The type of service to be offered dictates the design of a client–server application. The core of the client-side and server-side software is dependent on the communication between them, as outlined in the application protocol to be implemented. Such communication, barring a few services, is always based on a client sending a request to a server and the server sending a response to the client request. For some applications, such interaction is limited to one request and one response. However, for several well-known services, this request–response session continues until the client sends some sort of "quit" request to the server. When the server process has served a client, it is ready to accept a request from another client. If a single server process handles a client's requests, it is known as an *interactive server.*

Interactive servers can be connection oriented or connectionless. A connection-oriented service may be triggered simply by the presence of an incoming connection, without an explicit request from the client. Such a service is known as a *connection-triggered service* and the corresponding server as a *connection-triggered server.* For example, the well-known DAYTIME service is a connection-triggered service. A connection-oriented service may be based on serving a single request from a client. The server for such a service is known as a *one-shot, connection-oriented server.* TIME and ECHO are examples of such well-known services.

If a connection-oriented server has to respond to several requests from a client before moving to the next client, the iterative version of such a server will cause unnecessarily long delays at the client side. This would result in a long average waiting time for clients and, possibly, lost connection requests from clients if the queue associated with the server-side passive socket overflows. This necessitates the design of servers that can handle multiple clients simultaneously. Such servers are known as *connection-oriented, concurrent servers.* These servers use slave processes to handle multiple clients simultaneously. Several well-known services, including HTTP (WWW), FTP, SSH, and TELNET are offered through connection-oriented, concurrent servers.

This discussion leads us to the following types of servers:

1. Iterative connectionless
2. Iterative connection-oriented
    a. Connection-triggered
    b. Interaction based
3. Concurrent connectionless
4. Concurrent connection-oriented
    a. Master–slave-based
    b. Master–slave, multiservice

In case of the connection-oriented, concurrent, master–slave model, the main server process is known as the *master server process* and a process created to serve a client is known as the *slave process.* The master server accepts a client request, creates a new socket, forks the slave process, and the slave process services the client by communicating with it using the newly created socket. The slave process may also overwrite itself with the executable for the service using a call in the `exec()` family. The master–slave model may be scaled to handle multiple clients simultaneously.

A server may offer a service using both stream (SOCK _ STREAM) and datagram (SOCK _ DGRAM) styles of communication. Similarly, a server may offer multiple services. Such a server is known as a *multiservice server*. Lastly, a multiservice server may offer all of the services that it offers using the stream and datagram styles of communication.

20.8 ALGORITHMS AND EXAMPLES FOR SOCKET-BASED CLIENT–SERVER SOFTWARE

We now discuss the algorithms, system call graphs, client–server interaction sequences, and example source code for the client–server models based on the types of servers discussed in the previous section. Note that the call graphs show the sequence of system calls for socket-based I/O in the client and server processes. Library and/or system calls for performing I/O with standard devices and for performing other ancillary operations, including conversion from host to network byte order and vice versa, ASCII to integer conversion, and translation of an IP address from the DDN to binary are not included in these call graphs. The interaction sequences show how system calls between the client and server processes rendezvous. The call graphs for the client and server processes are shown in solid lines, and interaction sequences are displayed in dotted lines between the two processes.

20.8.1 Iterative Connectionless Client–Server Model

In this model, the client and server processes communicate using the connectionless style of communication based on the UDP protocol. The server process waits for a client request, forms a reply after receiving the request, sends the response to the client process, and goes back to wait for the next request from the same or another client. Figures 20.14 and 20.15 show the algorithms, system call graphs, and interaction sequences for this client–server model.

We now discuss the example of an iterative connectionless client–server model to deliver the current time in human-readable form. In this example, we implement the well-known TIME service, except that it is offered at an arbitrary port and not at the well-known port 37. We discuss the source code for both the client and server processes for the application.

Before discussing the protocol for the TIME service and the relevant source code, we need to understand how UNIX and the Internet maintain time. UNIX maintains time

Server process	**Client process**
1. Create a socket for `SOCK_DGRAM` style of communication 2. Bind an address to the socket 3. Receive a request from a client using the `recvfrom()` system call 4. Prepare a response and send it to the client using the `sendto()` system call 5. Go to Step 3	1. Create a socket for `SOCK_DGRAM` style of communication 2. Send a request to the server using the `sendto()` system call 3. Receive server's response using the `recvfrom()` system call and process it according to the protocol for the service 4. Close the socket 5. Exit

FIGURE 20.14 Algorithms for the client and server processes for the iterative connectionless client–server model.

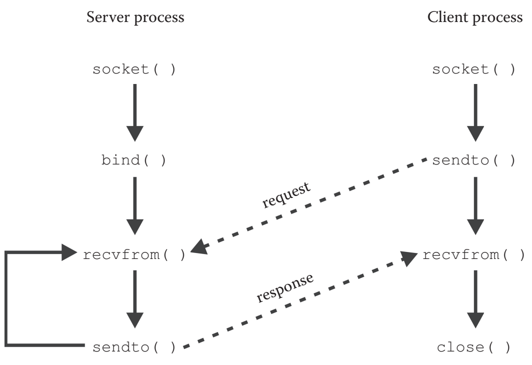


FIGURE 20.15 System call graphs and interaction sequences for the iterative connectionless client–server model.

in terms of the number of seconds since the *UNIX epoch*; that is, midnight, January 1, 1970. The Internet, on the other hand, maintains time in terms of the number of seconds since midnight, January 1, 1900. The number of seconds between these two baselines is 2,208,988,800 seconds. In other words, the UNIX epoch is 2,208,988,800 seconds away from the Internet baseline. Thus, if you use a function on a UNIX machine that returns time and you want to convert it to the Internet time, you need to add 2,208,988,800 to it. Conversely, if you receive time from the Internet and want to process it on a UNIX machine, you need to subtract 2,208,988,800 from it.

In the client–server model for the TIME service, the client process sends an arbitrary request to the server process. The server process, without even deciphering the client request, uses a function to get the current time with respect to the UNIX epoch, adds 2,208,988,800 to it, converts the resultant value (i.e., time in the Internet domain) to the network byte order, and sends it to the client process. The client process receives it, converts it from the network byte order to the host byte order, subtracts 2,208,988,800 from it, uses a function to convert it into human-readable form and displays the time, closes its socket, and quits. In the following and subsequent code examples, we define the symbolic constant UNIXEPOCH as 2,208,988,800.

Here is the source code for both the client and server software, their compilation, and a sample run. Note that we use the `connect()` system call in the client process. However, because the socket descriptor specified in the call is a UDP socket, no network traffic is therefore generated and no three-way handshake takes place between the client- and server-side sockets. Nonetheless, the address of the remote socket is stored in the local socket's data structure. This allows us to use the `read()` and `write()` system calls instead of `recvfrom()` and `sendto()` calls.

```
% more time_server_UDP.c
/*
Usage: Server-name Protocol Port
Here, Protocol is the transport level protocol
and Port is the procotol port number where the
```

```
service is to be offered.
*/
#include <time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define UNIXEPOCH 2208988800
#define QLEN      0
#define BUFF      256

int main(int argc, char *argv[])
{
    int s;

    /* Exit if program is not run with two command line
       arguments. */
    if (argc != 3) {
        printf("Usage: server protocol port\n");
        exit(1);
    }

    /* Create a TCP socket, bind name to it, and put it in passive
       mode */
    s = CreatePassiveSock(argv[1], argv[2], QLEN);

    /* Read client request and send current time to client */
    while (1) {
      /* UDP time service code */
      TimeServerUDP(s);
    }
}

int CreatePassiveSock(char *protocol, char *portstr, int qlen)
{
    /* Insert code for the function */
}

/* Provide TIME service to a client. 'sock' is the  */
/* server-side socket.  */
void TimeServerUDP(int sock)
{
    int n, saddrlen;
    char buf[BUFF];
    time_t current_time;
    struct sockaddr_in saddr;

    saddrlen = sizeof(saddr);
    /* Read client request and send current time to client */
    n = recvfrom(sock, buf, sizeof(buf), 0,
                 struct scokaddr *) &saddr, &saddrlen);
    if (n == -1) {
      perror("recvfom failed");
      exit(1);
    }
    (void) time(&current_time);
    current_time = htonl((u_long) (current_time + UNIXEPOCH));
    (void) sendto(sock, (char *) &current_time, sizeof(current_
                  time), 0,
                  struct sockaddr *)&saddr, saddrlen);
}
% more time_client_UDP.c
#include <time.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define UNIXEPOCH 2208988800
#define Message "Message to time sever\n"

int main(int argc, char *argv[])
{
    int csock; /* Descriptor for connected socket */

    if (argc != 4) {
        printf("Usage: %s hostname port protocol\n", argv[0]);
        exit(0);
    }

    csock = CreateConnectedSock(argv[1], argv[2], argv[3]);

    /* Perform client-side echo using csock, the connected
       socket */
    TimeClientUDP(csock);

    /* Deallocate socket and return */
    close(csock);
    return 0;
}

int CreateConnectedSock(char *ip, char *portstr, char *protocol)
{
    /* Insert code for the function */
}

/* Client side of the TIME service. 'sock' is the  */
/* socket "connected" to the server-side socket.   */
```

```
void TimeClientUDP(int sock)
{
    int n;
    time_t current_time;

    (void) write(sock, Message, strlen(Message));

    n = read(sock, (char *)&current_time, sizeof(current_time));
    if (n < 0) {
        perror("read failed");
        exit(1);
    }
    current_time = ntohl((u_long) current_time);
    current_time = current_time - UNIXEPOCH;
    printf("%s", ctime(&current_time));
}
%
```

Here are the compilation of the server and client software and a few sample runs. On a Solaris machine, you would use the `gcc –w -lsocket time _ server _ UDP.c -o udptimes` command to generate the executable code for the server program and the `gcc -w -lsocket -lnsl time _ client _ UDP.c -o udptimec` command to generate the executable code for the client program. As expected, execution of the client process with the TCP protocol failed.

```
% gcc46 time_server_UDP.c -w -o udptimes
% gcc46 time_client_UDP.c -w -o udptimec
% ./udptimes udp 6001 &
[1] 96361
% ps
  PID TT  STAT    TIME COMMAND
90518  1  Is+  0:00.54 -csh (csh)
93596  2  Is+  0:00.34 -csh (csh)
87939  3  Ss   0:00.79 -csh (csh)
96361  3  S    0:00.00 ./udptimes udp 6001
96362  3  R+   0:00.01 ps
% ./udptimec 127.0.0.1 6001 udp
Sat Aug 15 18:51:13 2015
% ./udptimec 127.0.0.1 6001 tcp
connect failed: Connection refused
% kill -9 96361
%
```

EXERCISE 20.14

Repeat this session on your UNIX system to verify that it works as expected.

EXERCISE 20.15

Run the client and server processes on different windows on the same machine and on different machines on the Internet.

20.8.2 Iterative Connection-Triggered Client–Server Model

In this model, the client and server processes communicate using the connection-oriented style of communication based on the TCP protocol. The server process waits for the connection request from a client process. As soon as it receives a connection request, it forms a reply without receiving an explicit request from the client process, sends the response to the client process, and goes back to wait for a connection request from another client. Figures 20.16 and 20.17 show the algorithms, system call graphs, and interaction sequences for this client–server model.

Server process	**Client process**
1. Create a socket for `SOCK_STREAM` style of communication 2. Bind an address to the socket using the `bind()` system call 3. Put the socket in passive mode using the `listen()` system call 4. Receive a connection request from a client using the `accept()` system call and communicate with the client using the newly created active socket 5. Prepare a response and send it to the client using the `write()` system call and close the active socket 6. Go to Step 4	1. Create a socket for `SOCK_STREAM` style of communication 2. Send a connection request to the server process using the `connect()` system call 3. Receive server's response using the `read()` system call and process it according to the protocol 4. Close the socket 5. Exit

FIGURE 20.16 Algorithms and system call sequences for the iterative connection-triggered client–server model.

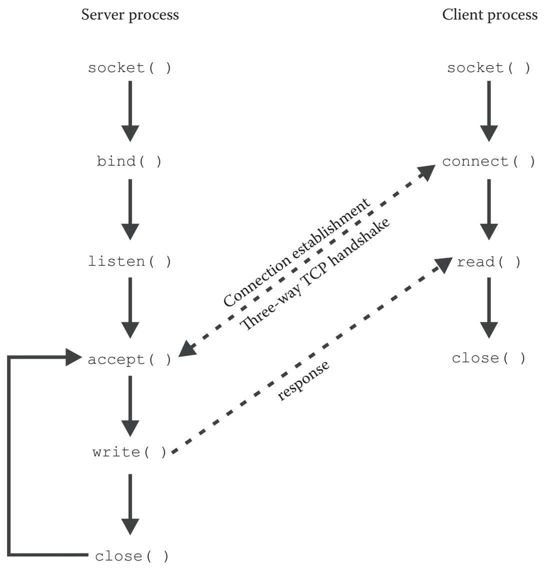


FIGURE 20.17 System call graphs and interaction sequences for the iterative connection-triggered client–server model.

We now discuss the example of an iterative connection-triggered client–server model to deliver the current time in human-readable form. In this example, we implement the well-known DAYTIME service, except that our service is offered at an arbitrary port and not at the well-known port 13. We discuss the source code for the client and server processes for the application.

The protocol for the client–server model for the TCP connection-triggered DAYTIME service is very similar to the TCP TIME service, except for two differences. First, the client process does not send any message to the server process after successfully establishing a connection with the server process. Second, whereas in the case of the TIME service, the current clock-tick count is converted into a human-readable form of time on the client side, in the case of the DAYTIME service, the conversion is carried out on the server side. The general structure of the client and server software is very similar to that of the TIME service, except, of course, for the functions to handle the current time on both sides.

Here is the source code for the client and server software:

```
% cat daytime_server_TCP.c
#include <stdio.h>
#include <time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define BUFF      256
#define QLEN      5

int main(int argc, char *argv[])
{
    int psock, asock, caddrlen;
    struct sockaddr_in caddr;

    /* Exit if program is not run with two command line
       arguments. */
    if (argc != 3) {
        printf("Usage: server protocol port\n");
        exit(1);
    }

    /* Create a TCP socket, bind name to it, and put it in passive
       mode */
    psock = CreatePassiveSock(argv[1], argv[2], QLEN);

    caddrlen = sizeof(caddr);
    while (1) {
      /* Accept actual connection from the client */
      if ((asock = accept(psock, (struct sockaddr *)&caddr,
                         &caddrlen)) == -1) {
          perror("accept failed");
```

```
            exit(1);
        }

        /* The Daytime service: Send current time as string to
           client */
        (void) TCPdaytime(asock);

        /* Deallocate the active socket and accept next
           connection */
        close(asock);
    } /* while */
}

int CreatePassiveSock(char *protocol, char *portstr, int qlen)
{
    /* Insert code for the function */
}

/* TCPdaytime(active socket descriptor) */
int TCPdaytime(int s)
{
    int     n;
    time_t  current_time;  /* Current time in ticks    */
    char    *str;          /* Pointer to time string   */
    char    *ctime();

    /* Get the current time in terms of clock ticks from    */
    /* UNIX Epoch, i.e., midnight January 1, 1970 and       */
    /* save it in current_time.                             */
    (void) time(&current_time);

    /* Convert current time into a humanly readable string */
    /* and return pointer to this string, saved in ptr     */
    str = ctime(&current_time);

    /* Send humanly readable time string to client process */
    if ((n = write(s, str, strlen(str))) == -1) {
        perror("write call failed");
        exit(1);
    }
    return 0;
}
% cat daytime_client_TCP.c
#include <netdb.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```
#define BUFF      256

int main(int argc, char *argv[])
{
    int csock; /* Descriptor for connected socket */

    if (argc != 4) {
      printf("Usage: %s hostname port protocol\n", argv[0]);
      exit(0);
    }

    csock = CreateConnectedSock(argv[1], argv[2], argv[3]);

    /* Perform client-side of the DAYTIME service using csock,  */
    /* the client socket connected the server-side active socket */
    DaytimeClientTCP(csock);

    /* Deallocate socket and return */
    close(csock);
    return 0;
}

int CreateConnectedSock(char *ip, char *portstr, char *protocol)
{
    /* Insert code for the function */
}

/* Client side of the DAYTIME service. 'sock' is the  */
/* client socket connected to serve's active socket.  */
void DaytimeClientTCP(int sock)
{
    int n;
    char buf[BUFF];

    /* Read time string from server and display on screen */
    while ((n = read(sock, buf, BUFF)) > 0)
      write (1, buf, n);
}
```

Here is the compilation and a sample run of the DAYTIME client–server software. As explained previously, on a Solaris machine you would use the `gcc –w -lsocket daytime _ server _ TCP.c -o udpdaytimes` command to generate the executable code for the server program and the `gcc -w -lsocket -lnsl daytime _ client _ TCP.c -o udpdaytimec` command to generate the executable code for the client program.

```
% gcc46 daytime_server_TCP.c -w -o tcpdaytimes
% gcc46 daytime_client_TCP.c -w -o tcpdaytimec
% ./tcpdaytimes tcp 6001 &
[1] 6566
% ps
 PID TT  STAT    TIME COMMAND
1291  1  Ss   0:00.66 -csh (csh)
6566  1  S    0:00.00 ./tcpdaytimes tcp 6001
6573  1  R+   0:00.01 ps
2963  4  Is+  0:00.42 -csh (csh)
% ./tcpdaytimec 127.0.0.1 6001 tcp
Sat Aug 15 20:58:22 2015
% kill -9 6566
%
```

EXERCISE 20.16

Repeat this session on your UNIX system to verify that it works as expected.

EXERCISE 20.17

Run the client and server processes in different windows on the same machine and on different machines on the Internet.

We have tried to structure our software from the three applications that we have discussed so far. However, we still see *code clones* in the client and server software, particularly in the `CreateConnectedSock()` and `CreatePassiveSock()` functions. This means that we can further improve the design of our client–server software by using *refactoring* and making functions out of these clones. We can further improve our design by using additional abstractions. These functions can be archived in a library on top of the existing libraries for network programming and system calls. We leave this work as an exercise for the reader.

20.8.3 Iterative One-Shot Connection-Oriented Client–Server Model

In this model, the client and server processes communicate using the connection-oriented style of communication based on the TCP protocol. The server process waits for the connection request from a client process, accepts a connection request from the client, forms a response after receiving an explicit request from the client process, sends the response to the client process, and goes back to wait for a connection request from another client. Figures 20.18 and 20.19 show the algorithms, system call graphs, and interaction sequences for this client–server model.

We now discuss the example of an iterative one-shot connection-oriented client–server model for the ECHO service. Our model implements the well-known ECHO service, except that it is offered at an arbitrary port and not at the well-known port 7. In this model for the ECHO service, the client process sends a connection request to the server process. The server process accepts the connection request, receives some text from the client process, sends back the same text to the client process, closes the active socket, and goes back to

Server process	Client process
1. Create a socket for `SOCK_STREAM` style of communication 2. Bind an address to the socket using the `bind()` system call 3. Put the socket in passive mode using the `listen()` system call 4. Receive a connection request from a client using the `accept()` system call and communicate with the client using the newly created active socket 5. Receive an explicit request from a client process 6. Prepare a response and send it to the client using the `write()` system call and close the active socket 7. Go to Step 4	1. Create a socket for `SOCK_STREAM` style of communication 2. Send a connection request to the server process using the `connect()` system call 3. Send a request to the server process using the `write()` system call 4. Receive server's response using the `read()` system call and process it according to the protocol 5. Close the socket 6. Exit

FIGURE 20.18 Algorithms and system call sequences for the iterative one-shot connection-oriented client–server model.

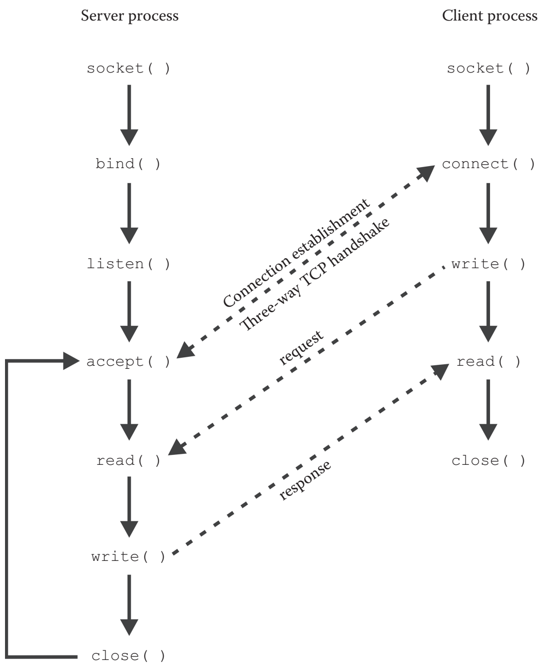


FIGURE 20.19 System call graphs and interaction sequences for the iterative one-shot connection-triggered client–server model.

accept another client request. The client process receives the text from the server process, displays it on the screen, closes its socket, and quits.

EXERCISE 20.18

Write the code for the iterative one-shot connection-oriented client–server model for the ECHO service. Compile and run the implementation on your UNIX system to verify that it works as expected. Show compiler commands for generating the executable codes for the client and server programs for both PC-BSD and Solaris machines.

EXERCISE 20.19

Run the client and server processes in different windows on the same machine and on different machines on the Internet.

20.8.4 Iterative Connection-Oriented Client–Server Model

In this model, the client and server processes communicate using the connection-oriented style of communication based on the TCP protocol. The server process waits for the connection request from a client process, accepts the connection request from a client process, and waits for a service request from the client. After receiving a request from the client, the server process forms a response, sends it to the client process, and waits for another request from the same client. The client process receives the response, processes it according to the underlying algorithm, and sends the next request to the server process. This interaction between the client and server processes continues until the client sends some sort of "quit" request to the server process. On receiving this request, the server and client processes disconnect gracefully by closing their respective sockets in an orderly manner. Figures 20.20 and 20.21 show the algorithms, system call graphs, and interactions sequences for this client–server model.

We now discuss an example of an iterative connection-oriented client–server model that deals with three requests from a client: "echo", "daytime", and "quit". A client process may repeat each of these requests multiple times. The client takes requests from the user, accepts a request only if it is one of these three, sends it to the server process, and waits for the server response. For any other request, the client process displays an error message on the screen and prompts the user for another request. If the user input is "echo",

Server process	**Client process**
1. Create a socket for `SOCK_STREAM` style of communication 2. Bind an address to the socket using the `bind()` system call 3. Put the socket in passive mode using the `listen()` system call 4. Receive a connection request from a client using the `accept()` system call and communicate with the client using the newly created active socket 5. Receive an explicit request from the client process using the `read()` system call 6. If the received request is some kind of "quit", then go to Step 9 7. Prepare a response and send it to the client process using the `write()` system call 8. Go to Step 5 9. Close the active socket created by the `accept()` system call and go to Step 4	1. Create a socket for `SOCK_STREAM` style of communication 2. Send a connection request to the server process using the `connect()` system call 3. Send an explicit request to the server process using the `write()` system call 4. Receive server's response using the `read()` system call and process it according to the service protocol 5. If the request sent was not some sort of "quit", then go to Step 3 6. Close the socket 7. Exit

FIGURE 20.20 Algorithms and system call sequences for the iterative connection-oriented client–server model.

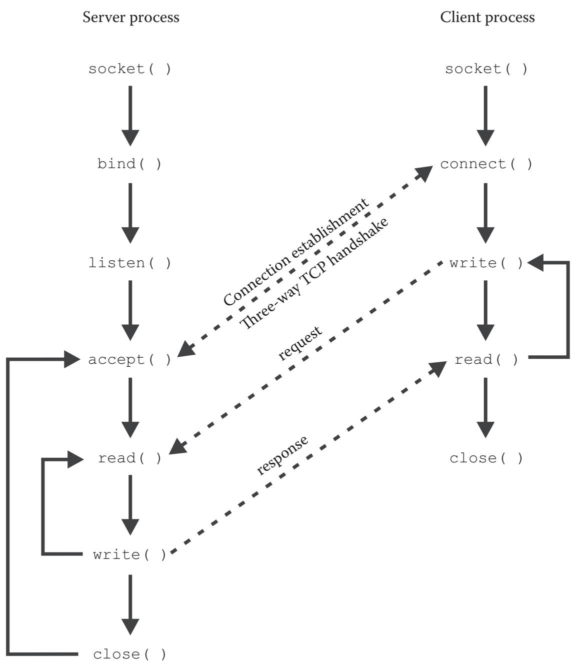


FIGURE 20.21 System call sequences and interaction sequences for the general iterative connection-oriented client–server model.

the model implements the well-known ECHO service. If the user input is "daytime", the model implements the DAYTIME service. If the user input is "quit", the client and server processes disconnect gracefully. The service is offered at an arbitrary port.

EXERCISE 20.20

Write the code for the iterative connection-oriented client–server model for the ECHO service. Compile and run the implementation on your UNIX system to verify that it works as expected.

EXERCISE 20.21

Run the client and server processes in different windows on the same machine and on different machines on the Internet.

20.8.5 Concurrent Connectionless Client–Server Model

In this model, the client and server processes communicate using the connectionless style of communication based on the UDP protocol. The master server process waits for a request from a client process, creates a slave process, hands over the client request and socket to the slave process, and goes back to receive another client request. The slave process forms a response according to the application protocol, sends it to the client process, and exits. Figures 20.22 and 20.23 show the algorithms, system call graphs, and interaction

Server process

Master process

1. Create a socket for `SOCK_DGRAM` style of communication
2. Bind an address to the socket using the `bind()` system call
3. Receive a request from a client using the `recvfrom()` system call and create a slave process using the `fork()` system call to handle the client request
4. Go to Step 3

Slave process

1. Receive the client request as well as access to the socket
2. Prepare a response according to the application protocol and send it to the client process using the `sendto()` system call
3. Exit after serving a request

FIGURE 20.22 Algorithm for the concurrent connectionless server process.

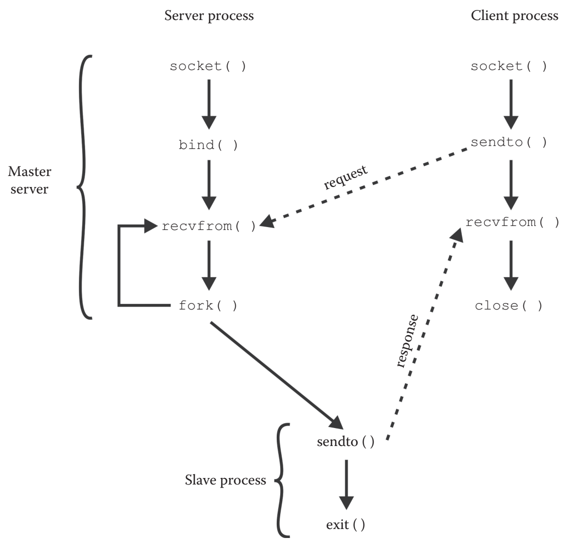


FIGURE 20.23 System call graphs and interaction sequences for the concurrent connectionless client–server model.

sequences for this client–server model. Note that the algorithm for the client process is the same as for an iterative connectionless server discussed in Section 20.7.1.

20.8.6 Concurrent Connection-Oriented Client–Server Model

In this model, the client and server processes communicate using the connection-oriented style of communication based on the TCP protocol. The server process waits for the connection request from a client process, accepts the connection request, creates a slave process, hands over the connection and socket to the slave process, and goes back to accept another connection request. The slave process receives an explicit request form the connected client process, forms a response according to the application protocol, sends the response to the client process, and waits for the next request from the client process. This request–response session continues until the client sends a "quit" request of some sort. On receiving the "quit" request, the slave and client processes disconnect gracefully and close their sockets.

Server process

Master process

1. Create a socket for SOCK_STREAM style of communication
2. Bind an address to the socket using the bind() system call
3. Put the socket in passive mode using the listen() system call
4. Receive a connection request from a client using the accept() system call and create a slave process using the fork() system call to handle the request.
5. Go to Step 4

Slave process

1. Receive the socket that was created by the accept() system call and is connected to the client-side socket
2. Receive a request from the client using the read() system call
3. If the received request is some kind of "quit", then go to Step 5
4. Prepare a response according to the application protocol, send it to the client using the write() system call, and go to Step 2
5. Close the active socket created by the accept() system call, disconnect with the client process gracefully, and exit

FIGURE 20.24 Algorithm for the concurrent connection-oriented server process.

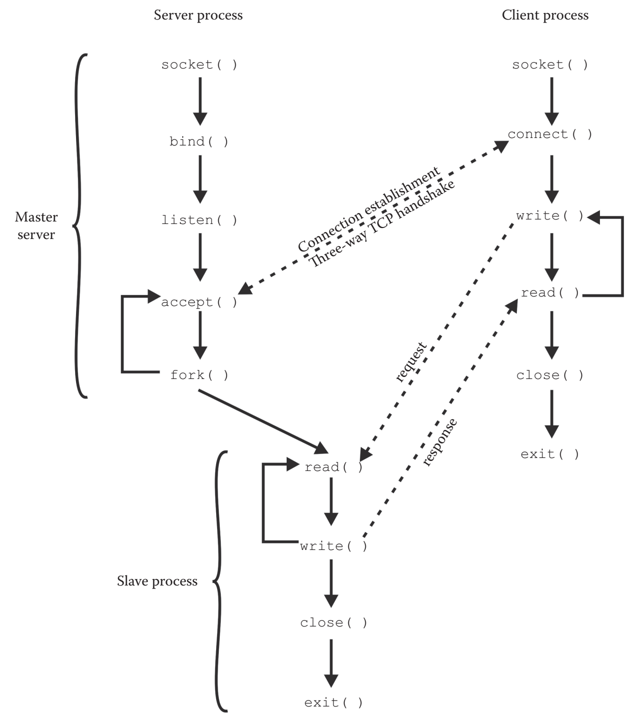


FIGURE 20.25 System call graphs and interaction sequences for the concurrent connection-oriented client–server model using slave processes.

Figures 20.24 and 20.25 show the algorithms, system call graphs, and interaction sequences for this client–server model. Note that the algorithm for the client process is the same as for an iterative connectionless server as discussed in Sections 20.6.12 and 20.8.1.

Figure 20.26 shows the pictorial view of the concurrent connection-oriented client–server model with *k* clients being handled by *k* slave processes and the master process waiting to accept a connection request from another client process.

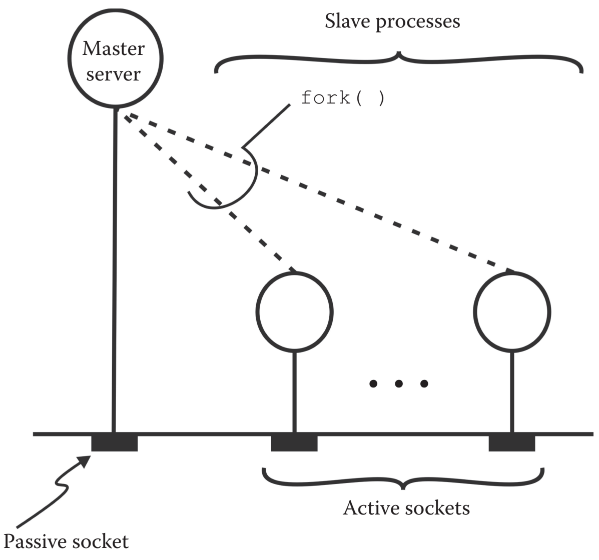


FIGURE 20.26 Connection-oriented concurrent server.

Here is the source code for the concurrent connection-oriented server software based on slave processes for the DAYTIME service. We have replaced the call to the `TCPdaytime()` function in the `while()` loop to a piece of code for creating a child (slave) process that closes the passive socket and calls the `TCPdaytime()` function to handle the client request. The parent process closes the active socket and goes back to accept the next connection from a client process.

```
% cat daytime_concurrent_server_TCP.c
#include <stdio.h>
#include <time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define BUFF      256
#define PORT      6001
#define QLEN      5

int main(int argc, char *argv[])
{
    int asock, psock, pid, caddrlen;
    struct sockaddr_in caddr;

    /* Exit if program is not run with two command line
       arguments. */
    if (argc != 3) {
        printf("Usage: server protocol port\n");
        exit(1);
    }

    /* Create a TCP socket, bind name to it, and put it in passive
       mode */
    psock = CreatePassiveSock(argv[1], argv[2], QLEN);

    while (1) {
      /* Accept actual connection from the client */
      if ((asock = accept(psock, (struct sockaddr *)&caddr,
      &caddrlen)) == -1) {
          perror("accept failed");
          exit(1);
      }
      /* Create a slave process and have it server the client    */
      /* The parent process closes the newly created active      */
      /* socket and goes back to accept the next client request */
      pid = fork();
      if (pid == -1) {
          perror("fork failed");
          exit(1);
      }
      if (pid == 0) { /* Child process */
          close(psock); /* Deallocate passive socket */
          /* The Daytime service: Send current time as string */
          /* to client using active socket                    */
          (void) TCPdaytime(asock);
          exit(0);
      }
      else /* Parent process: Deallocate the active socket      */
          close(asock);
    } /* while */
}

int CreatePassiveSock(char *protocol, char *portstr, int qlen)
{
    /* Insert code for the function */
}

int TCPdaytime(int s)
{
    /* Insert code for the function */
}
```

Here is the compilation of the source, the running of the resultant executable code, and its testing with the DAYTIME client developed in Section 20.8.2. The outputs of the various commands are self-explanatory. The session shows that you can use the loopback address with the client command instead of explicitly specifying the IP address of the host on which the client and server processes run.

```
% gcc46 daytime_concurrent_server_TCP.c -w -o tcpdaytimecons
% ./tcpdaytimecons tcp 6001 &
[1] 18738
% ./tcpdaytimec 202.147.169.196 6001 tcp
Sat Aug 15 22:56:46 2015
% ps
  PID TT  STAT    TIME COMMAND
 1291  1  Is+  0:00.73 -csh (csh)
16351  2  Ss   0:00.46 -csh (csh)
18738  2  S    0:00.00 ./tcpdaytimecons tcp 6001
18822  2  Z    0:00.00 <defunct>
18827  2  R+   0:00.01 ps
16380  3  Is+  0:00.16 -csh (csh)
% ./tcpdaytimec 127.0.0.1 6001 tcp
Sat Aug 15 22:56:55 2015
%
  PID TT  STAT    TIME COMMAND
 1291  1  Is+  0:00.73 -csh (csh)
16351  2  Ss   0:00.47 -csh (csh)
18738  2  S    0:00.00 ./tcpdaytimecons tcp 6001
18822  2  Z    0:00.00 <defunct>
18840  2  Z    0:00.00 <defunct>
18849  2  R+   0:00.01 ps
16380  3  Is+  0:00.16 -csh (csh)
% kill -9 18738 18822 18840
%
```

EXERCISE 20.22

Repeat this session on your UNIX system to verify that it works as expected. Show the compiler commands to generate the executable codes for the client and server programs on a Solaris machine.

EXERCISE 20.23

Run the client and server processes in different windows on the same machine and on different machines on the Internet. Verify working of the concurrent server with multiple simultaneous clients.

Note that the outputs of the `ps` commands show that the child/slave processes become *zombies* when they terminate. This happens because its parent is not waiting when it terminates. It seems that an obvious solution for this problem is to use the `wait()` (or a variant of this call) before the `close(sock)` statement in the parent's code. However, this will

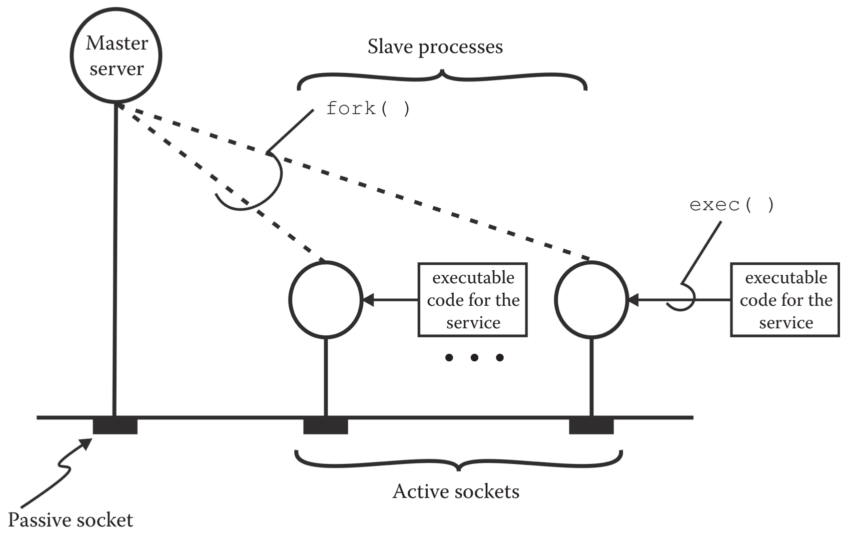


FIGURE 20.27 Concurrent connection-oriented server process using `fork()` and `exec()`.

make the parent process wait until the child (slave) process terminates, thereby making the model iterative. The real solution for the problem is to let the child become a zombie and then immediately remove it from the system. This can be achieved via the use of signal handling, as discussed in Section 20.5. Recall that when a child process terminates, the UNIX kernel generates a `SIGCHLD` signal. You can insert a few lines of code in the server process to intercept this signal and run the `wait3()` system call to remove the relevant zombie from the system. We discuss this issue in detail in Section 21.12. For now, you can either remove zombies manually by using the `kill` command or use the code segments given in Section 21.12 to clean up a zombie process as it is created. For now, we use the `kill` command to remove zombie processes from the system.

The structure of the server process in this model may be extended by having each slave process overwrite itself with another executable file using a call from the `exec()` family. The executable file corresponds to the service to be provided for a given client request. A pictorial view of such a server process is shown in Figure 20.27.

20.9 SYNCHRONOUS VERSUS ASYNCHRONOUS I/O: THE `SELECT()` SYSTEM CALL

UNIX system calls such as `read()`, `write()`, `recvfrom()`, `sendto()`, and `accept()` block if the socket descriptors associated with them are not ready to perform input, output, or connection establishment [in the case of `accept()`]. As soon as the socket descriptors associated with any of these calls are ready for I/O or connection establishment, the calls unblock and perform their designated operation.

The I/O based on signals also works in a similar manner. As soon as a signal occurs, the associated signal-handling code executes. The I/O based on blocking calls and signals is known as *synchronous I/O*. Both work similarly to interrupt handling in a computer system.

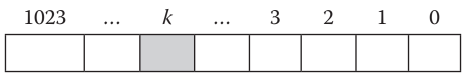


FIGURE 20.28 Pictorial view of a descriptor set.

There are times when you need to perform nonblocking, asynchronous I/O and connection establishment. You may perform *asynchronous I/O* using the `select()` system call. Before we discuss the algorithm for the connection-oriented concurrent server that works based on the principles of asynchronous I/O, we need to discuss the `select()` system call. It is a powerful system call that, depending on the value of one of its parameters, allows you to perform synchronous as well as asynchronous I/O.

The `select()` system call deals with descriptor sets. A *descriptor set* is a bit mask in which a bit represents a descriptor and its value indicates the state of the corresponding descriptor. The size of the bit mask is defined in **/usr/include/sys/select.h** as `FD _ SETSIZE`. This value is usually at least as large as the number of descriptors in the PPFDT on the system. On our BSD and Solaris systems, it is defined as unsigned 1024. Descriptor numbers and bit numbers start with 0. Thus, bit *k* in a descriptor set represents descriptor number *k*, as shown in Figure 20.28.

The `select()` call takes three descriptor sets as arguments, one each for reading, writing, and exceptions. This call is normally used for socket I/O. Thus, the descriptors in the sets are usually socket descriptors. The call monitors the descriptors in these sets for input (reading), output (writing), and exceptions. If a bit in a descriptor set has a value of 0 (i.e., not set), it means that the corresponding descriptor is not ready for reading, writing, or exception, depending on which descriptor set in the `select()` call we are referring to. If a bit has a value of 1 (i.e., it is set), this means that the corresponding descriptor is ready for the purpose for which it is intended—input, output, or exception.

Here is a brief description of the `select()` system call.

```
#include <sys/select.h>
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

Success: Number of ready descriptors out of those specified in the descriptor sets;
0 if time limit expires and no descriptor is ready

Failure: `-1` and kernel variable `errno` set to indicate the type of error; descriptor sets remain unmodified

Here, `nfds` is the number of file descriptors in the descriptor sets that need to be monitored, `readfds` is the read descriptor set, `writefds` is the set of descriptors on which the write operation is possible, `exceptfds` is the set of descriptors on which exceptions are possible, and `timeout` is the time for which the `select()` call waits for the selection process to complete. If you are not interested in one or more items in the descriptor set, you may specify them as null pointers. The `fd _ set` type is defined as `unsigned long` in **/usr/include/sys/select.h**. Thus, descriptor sets are stored as bit fields in arrays of unsigned long. As stated in Table 20.2, the following macros are used to manipulate descriptor sets: `FD_ZERO()`, `FD_SET()`, `FD_CLR()`, and `FD_ISSET()`. Table 20.21 gives the syntax and semantics of these macros. The behavior of these macros is undefined if `fd` is greater than `FD_SETSIZE` or less than 0.

The amount of time that `select()` monitors the descriptor sets is dependent on the value of the `timeout` argument, as shown in Table 20.22.

The `timeval` structure is defined in the **/usr/include/sys/time.h** file as shown next. Thus, a variable of the `timeval` structure may be used to specify time with microsecond granularity.

```
struct timeval {
    time_t       tv_sec;    /* seconds      */
    suseconds_t  tv_usec;   /* microseconds */
};
```

The `select()` call may fail for several reasons. Table 20.23 lists some of the reasons why the `accept()` call may fail.

The following program illustrates the semantics of the `select()` system call.

Figure 20.29 shows the algorithm for a single-process connection-oriented concurrent server using the `select()` system call. Note that this server serves one client request at a time and then moves to serve the request from the next client.

We now discuss the single-process connectionless concurrent server for the TIME and ECHO services using the `select()` system call. The following session contains the code

TABLE 20.21 Macros for Manipulating Descriptor Sets

Macro	Purpose
`FD_ZERO(&fdset)`	Initializes the `fdset` descriptor set to the null set (i.e., all zeros)
`FD_SET(fd, &fdset)`	Includes the given descriptor in the set (i.e., sets the relevant numbered bit to 1)
`FD_CLR(fd, &fdset)`	Excludes the given descriptor from the set (i.e., sets the relevant numbered bit to 0)
`FD_ISSET(fd, &fdset)`	Tests if `fd` in `fdset` has a value of 0 or 1. Returns 0 if `fd` is not set in `fdset`, nonzero otherwise. It is usually used after the `select()` call has returned to see if the given descriptor is ready for I/O or exception

TABLE 20.22 The Value of the `timeout` Argument and Its Effect

Value of `timeout`	Effect
Not a null pointer	The value in the variable of `struct timeval` type specifies the maximum time for the selection process to complete
A null pointer	The `select()` call blocks indefinitely until a descriptor in a descriptor set is ready for the activity it is designated for (input, output, or exception)
A pointer to a zero-valued `timeval` structure	The `select()` call continuously polls the descriptors in the descriptor sets to determine if any is ready

TABLE 20.23 Reasons for the `select()` Call to Fail

Reason for Failure	**Value of** `errno`
One of the descriptor sets contains an invalid descriptor	`EBADF`
One (or more) of the following arguments points to an illegal address: `readfds`, `writefds`, `exceptfds`, or `timeout`	`EFAULT`
A signal occurred before the time limit expired and before any of the descriptors became ready	`EINTR`
`timeout` is invalid (i.e., one of the fields of this argument is too big or negative)	`EINVAL`
`nfds` is invalid	`EINVAL`

Server process

1. Create a socket for `SOCK_STREAM` style of communication.
2. Bind an address to the socket using the `bind()` system call.
3. Put the socket in passive mode using the `listen()` system call.
4. Add the socket descriptor to the set of those sockets on which I/O or exception is possible—to keep the discussion simple, we will not deal with sockets meant to monitor exception conditions.
5. Use the `select()` system call to monitor and select those descriptors that are ready for I/O.
6. See which of the sockets are ready by checking the descriptors in the descriptor sets. If the passive socket is ready, use the `accept()` system call to accept the connection request from a client and add the newly created socket to the set of those on which I/O is possible.
7. If a socket other than the passive socket is ready, receive the next request from an already connected client using the `read()` system call, form a response, and send it to the client using the `write()` system call.
8. If the client request is to close a connection, close the connection gracefully, set the corresponding descriptor value to 0 in the original descriptor set on which I/O was possible.
9. Set the 'readfds,' 'writefds,' and 'exceptfds' descriptor sets to the original set of descriptors on which I/O is possible (after accommodating any changes outlined in Step 8)
10. Go to Step 5.

FIGURE 20.29 Algorithm for the single-process connection-oriented concurrent server process using the `select()` system call.

for the server, its compilation, execution, and testing using the TIME and ECHO client discussed in Section 20.8.2.

```
% more select_server.c
#include <time.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/select.h>

#define UNIXEPOCH 2208988800
#define TIME      "8001"
#define ECHO      "8002"
#define BUFF      256
#define QLEN      5
```

```
int main(int argc, char *argv[])
{
    int s1, s2, nfds;
    fd_set rfds;

    /* Create UDP sockets for the TIME and ECHO services at   */
    /* ports 8001 and 8002, respectively, bind names to them, */
    /* and return their descriptors in s1 and s2.             */
    s1 = CreatePassiveSock("udp", TIME, QLEN);
    s2 = CreatePassiveSock("udp", ECHO, QLEN);

    /* Set number of descriptors to be monitored by select   */
    nfds = s2+1;

    /* Set the read descriptor set to zero                   */
    FD_ZERO(&rfds);

    /* Set bits for TIME and ECHO in the read descriptor set */
    FD_SET(s1, &rfds);
    FD_SET(s2, &rfds);

    while (1) {
      if (select(nfds, &rfds, NULL, NULL, NULL) == -1) {
          perror("select failed");
          exit(1);
      }
      if (FD_ISSET(s1, &rfds)) {
          /* UDP time service code */
          TimeServerUDP(s1);
      }
      if (FD_ISSET(s2, &rfds)) {
          /* UDP time service code */
          EchoServerUDP(s2);
      }
      /* Reset descriptors */
      FD_SET(s1, &rfds);
      FD_SET(s2, &rfds);
    }
}

int CreatePassiveSock(char *protocol, char *portstr, int qlen)
{
    /* Insert code for the function */
}

void TimeServerUDP(int sock)
{
    /* Insert code for the function */
}
```

```
/* Provide echo service to a UDP client.                         */
void EchoServerUDP(int sock)
{
    int n, caddrlen;
    char buf[BUFF];
    struct sockaddr_in caddr;

    /* Communicate with client: read and write back              */
    memset(buf, 0, BUFF);
    n = recvfrom(sock, buf, sizeof(buf), 0,
                (struct sockaddr *)&caddr, &caddrlen);
    if (n == -1) {
      perror("recvfom failed");
      exit(1);
    }
    (void) sendto(sock, buf, sizeof(buf), 0,
                  struct sockaddr *)&caddr, caddrlen);
}
%
```

The program for the server process is fairly straightforward and similar to the code previously written in this chapter, except for the code related to the `select()` system call. We create two UDP sockets, one each for the two services, and bind addresses to them using the `CreatePassiveSock()` function. We then initialize the `nfds` variable to the number of bits to be monitored in the read descriptor set, `rfds`, initialize the read file descriptor set to 0, and the set bits in `rfds` corresponding to the two socket descriptors. This is followed by the `select()` system call with 0 (i.e., `NULL`) values for the write descriptor set, exception descriptor set, and `timeval` structure. The value of 0 for the `timeval` structure means that the `select()` call blocks indefinitely until a descriptor becomes ready. When a client request arrives at either or both sockets, then `select()` returns. Depending on the bit(s) in `rfds` that are set, the corresponding client is served. After serving a client request, we reset the bits in `rfds`. Note that we have written a new function, `EchoServerUDP()`, to serve UDP clients for the ECHO service.

Here is a sample run of our client–server model. We test the two services by using the UDP TIME client and the TCP ECHO client. The TCP ECHO client works for the UDP service also, because we run a connect on the UDP socket in our code. We use the `udpechoc` and `udptimec` clients developed in Section 20.6.12 and 20.8.1, respectively.

```
% gcc46 select_server.c -w -o selects
% ./selects &
[1] 67008
% ps
  PID TT  STAT    TIME COMMAND
59200  3  Is+  0:00.81 -csh (csh)
59284  4  Ss   0:00.97 -csh (csh)
```

```
67008  4  S    0:00.00 ./selects
67009  4  R+   0:00.00 ps
% ./udptimec 202.147.169.196 8001 udp
Wed Aug 19 00:11:09 2015
% ./udptimec 127.0.0.1 8001 udp
Wed Aug 19 00:11:13 2015
% ./udpechoc 202.147.169.196 8002 udp
Enter text for server : That's all folks!
Text from server : That's all folks!
% kill -9 67008
%
```

EXERCISE 20.24

Repeat this session on your UNIX system to verify that it works as expected.

EXERCISE 20.25

At which port is each service offered?

EXERCISE 20.26

Run the client and server processes in different windows on the same machine and on different machines on the Internet. Verify working of the concurrent server with multiple simultaneous clients.

20.10 THE UNIX SUPERSERVER: INETD

When you offer a network service on a system, the corresponding daemon runs on the system. This means that the main memory and several kernel data structures allocated to the daemon are used. If some services are sparingly used, you can offer them through a single server that monitors the client requests on the sockets associated with the services and runs the corresponding server daemons only on demand. This scheme makes efficient use of the main memory and kernel data structures. A server that offers multiple network services is called a *superserver.*

In UNIX, `inetd` offers several basic Internet services and is known as the *UNIX superserver.* It is *dynamically configurable*; that is, it can reconfigure itself while it runs. It executes as a single master process, creates sockets for each of the services that it is to offer according to the type of communication (`SOCK _ STREAM`, `SOCK _ DGRAM`, etc.) and relevant protocol to be used (TCP, UDP, etc.), binds a name to each socket, and monitors client requests on these sockets using the `select()` system call. When a client request arrives on a socket, the master server process forks a slave (child) process, hands over the request to it, and goes back to wait for new client requests. The slave process uses a call in the `exec()` family to overlay itself with the executable code for the corresponding service. The **/etc/inetd.conf** file specifies complete information about all the services that `inetd` offers, one line per service, as discussed in the next subsection. Figure 20.30 shows the basic setup for `inetd` as it starts running. Figure 20.31 shows the state of `inetd` while it handles one UDP client for the TELNET service, two requests from TCP clients for the ECHO service, and one TCP client each for the TIME, TELNET, and FINGER services.

As an administrator (i.e., superuser), you should offer only lightly loaded services through this server. For example, you should offer SSH as a stand-alone service and the TELNET service through `inetd`, because most users today use the SSH service and TELNET is rarely used. When you offer a service through `inetd` and it becomes heavily loaded, then the performance

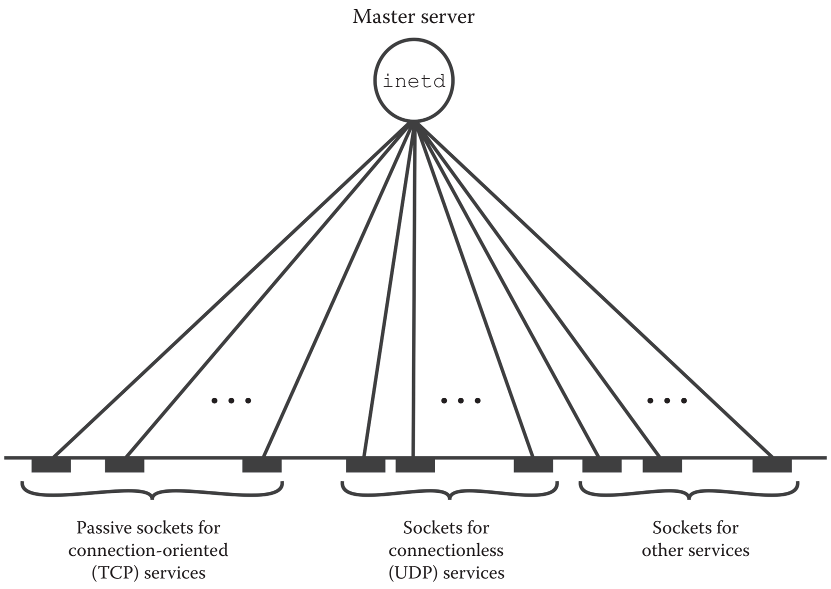

Master server
inetd
Passive sockets for connection-oriented (TCP) services
Sockets for connectionless (UDP) services
Sockets for other services

FIGURE 20.30 Basic setup of `inetd` as it starts running to offer TCP-based services.

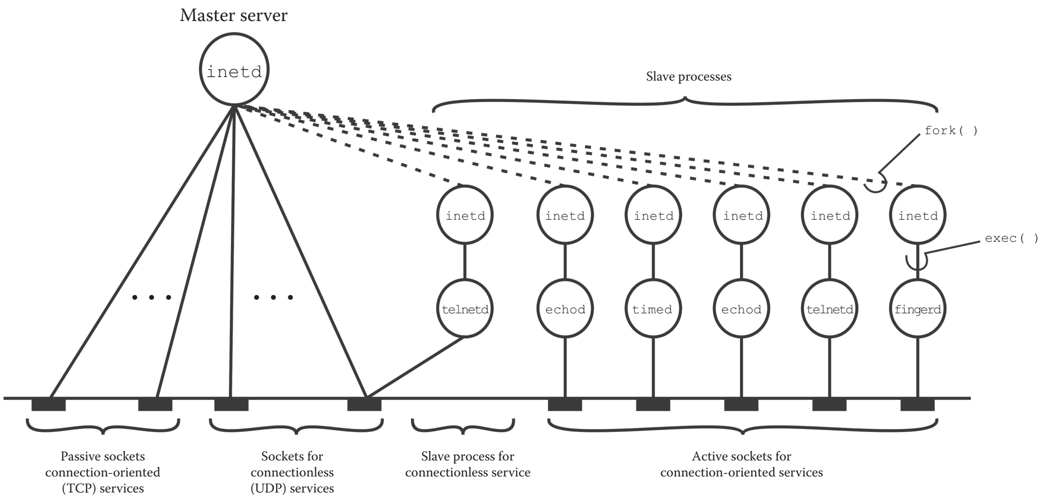

Master server
inetd
Slave processes
fork( )
exec( )
inetd
telnetd
echod
timed
echod
telnetd
fingerd
Passive sockets connection-oriented (TCP) services
Sockets for connectionless (UDP) services
Slave process for connectionless service
Active sockets for connection-oriented services

FIGURE 20.31 Setup of `inetd` as it serves two connection-oriented ECHO clients, one connectionless TELNET client, and one connection-oriented client each for TIME, TELNET, and FINGER.

of other services under inetd deteriorates. This is so because, with a higher number of slave processes serving the clients for a particular service, inetd has to multiplex a greater number of sockets associated with the slave processes. This means longer average waiting times for other services. Under these scenarios, you should remove such services from inetd.

We discuss briefly how inetd works, the kind of services it offers, and how you can change the set of services that it offers. On some UNIX systems, the *tcpwrapper* package is used to enhance the security of inetd, and on other UNIX systems this functionality is built into inetd.

20.10.1 Managing inetd on Solaris via Service Management Facility

The inetd daemon is invoked at system start-up time, and is configured via Oracle Solaris service management facility (SMF). We give a brief description and usage example of SMF with respect to inetd, particularly the svcadm command and its options in Sections 23.2.5.4, 23.2.5.4.2, and 23.2.5.5.

The SMF framework manages system and application services. SMF manages critical system services essential to the working operation of the system and manages application services such as databases or Web servers. SMF improves the availability of a system by ensuring that essential system and application services run continuously, even in the event of hardware or software failures.

SMF replaces the use of configuration files for managing services and is the recommended mechanism to use to start applications. SMF replaces the init scripting start-up mechanism, **inetd.conf** configurations, and most rc.d scripts. SMF preserves compatibility with existing administrative practices wherever possible.

The SMF command line utility that controls and configures inetd services is named inetadm. The inetadm command is used to monitor, configure, and control inetd services. If you type inetadm on the command line, you get a complete listing of inetd services, their properties, and values.

The contents of the file **/etc/services** lists the ports that the network services use. It would be very instructive for you to scroll through the content of the **/etc/services** file to learn what services are provided on what ports on your Solaris system.

See the inetd and inetadm manual page in Solaris for more details on how to manage inetd services. If necessary, you can use the inetconv command to convert an inetd configuration file content into SMF format services, and then manage these services using inetadm and svcadm commands.

20.10.1.1 Configuring inetd on PC-BSD

The **/etc/inetd.conf** file contains the list of services offered by inetd on a UNIX system. It is a text file that contains one seven-field line per service, having the following format:

```
service-name socket-type protocol wait/nowait user:group server-
program arguments
```

Each of the services listed in **/etc/inetd.conf** must also be recorded in the **/etc/services** file. Table 20.24 contains the purpose of each field.

TABLE 20.24 Fields of a Line in /etc/inetd.conf and Their Purpose

Field	Purpose
Service name	A port number in decimal or name corresponding to the service as listed in the /etc/services file
Socket type	The type of socket that the service is offered for, usually, `stream` for TCP style communication or `dgram` for UDP style communication. It could also be `raw`, `rdm`, or `seqpacket`
Protocol	For TCP, this field is `tcp` or `tcp6` for IPv4 and IPv6, respectively. For UDP, this field is `udp` or `udp6` for IPv4 and IPv6, respectively. For protocols based on remote procedure call (RPC), this field may be `rpc/tcp` or `rpc/udp`
Wait/nowait	This field specifies if `inetd` should wait until the service program terminates (`wait`) or continue (`nowait`). For `stream` type sockets, this field is normally `nowait`
User:group	The user name and optional group name that the respective service process runs as. Most of the services run under root ownership
Server-program	Absolute pathname of the program that is executed for the offered service
Arguments	This field contains the name of the program and the arguments with which it runs

Here are the contents of the **/etc/inetd.conf** file on our BSD system.

```
% cat /etc/inetd.conf
# $FreeBSD$
#
# Internet server configuration database
#
# Define *both* IPv4 and IPv6 entries for dual-stack support.
# To disable a service, comment it out by prefixing the line
with '#'.
# To enable a service, remove the '#' at the beginning of the
line.
#
ftp      stream  tcp   nowait root /usr/libexec/ftpd     ftpd -l
ftp      stream  tcp6  nowait root /usr/libexec/ftpd     ftpd -l
ssh      stream  tcp   nowait root /usr/sbin/sshd        sshd -i -4
ssh      stream  tcp6  nowait root /usr/sbin/sshd        sshd -i -6
telnet   stream  tcp   nowait root /usr/libexec/telnetd  telnetd
telnet   stream  tcp6  nowait root /usr/libexec/telnetd  telnetd
... [output truncated]
%
```

Note that the service name in the first field is different from the program name given in the last field, which is the name of the program (daemon), along with command line arguments, that executes when `inetd` receives a client request for this service. Any user may display the contents of this file.

Some of the services offered by `inetd` are labeled as “internal.” The following session shows such services on our BSD system.

```
% grep "internal"$ /etc/inetd.conf
daytime stream   tcp    nowait     root    internal
daytime stream   tcp6   nowait     root    internal
daytime dgram    udp    wait       root    internal
daytime dgram    udp6   wait       root    internal
time    stream   tcp    nowait     root    internal
time    stream   tcp6   nowait     root    internal
time    dgram    udp    wait       root    internal
time    dgram    udp6   wait       root    internal
echo    stream   tcp    nowait     root    internal
echo    stream   tcp6   nowait     root    internal
echo    dgram    udp    wait       root    internal
echo    dgram    udp6   wait       root    internal
... [output truncated]
%
```

These services are quite trivial and are handled directly by `inetd` without running any external program (daemon). They are useful for testing purposes, but are prone to "denial of service" (DoS) attacks. Accordingly, you should disable them by commenting them out. You can list the services that are enabled by using the `grep -v "^#" /etc/inetd.conf` command.

20.10.1.2 Locating inetd Services on PC-BSD and Solaris

The **/etc/services** file is the database that contains the name of a service, the port number for the service, and the transport-layer protocol it uses for communication (usually `tcp` or `udp`). Any service that `inetd` offers must have an entry in the **/etc/services** file. It has one line per service in the following format:

```
service-name port-number/protocol-name [aliases]
```

Here, `service-name` is the name of the service such as `smtp`, `port-number` is the IANA assigned port number, such as `25` for `smtp`, and `protocol-name` is the transport-layer protocol, such as `tcp`, `udp`, and so on. Here is an example line from the **/etc/services** file:

```
pop3  110/tcp     #Post Office Protocol - Version 3
```

This line shows that `pop3` is a TCP (i.e., connection-oriented) service, offered at port 110.

The following session shows that there are three entries for the SSH service in the **/etc/services** file on our PC-BSD system. This means that the SSH service is offered on port 22 under the `tcp`, `udp`, and `sctp` protocols. Stream Control Transmission Protocol (`SCTP`) has been designed to transmit multiple streams of data between two connected sockets. It is also known as the "next generation TCP" or "TCPng." The SSHELL service on port 614 is for secure socket layer shell (SSLshell).

```
% grep ^ssh /etc/services
ssh     22/sctp #Secure Shell Login
ssh     22/tcp  #Secure Shell Login
ssh     22/udp  #Secure Shell Login
sshell  614/tcp #SSLshell
sshell  614/udp
%
```

20.10.1.3 Locating inetd Protocols on PC-BSD and Solaris

Another file that `inetd` reads when it runs is **/etc/services**. This file is the database of protocol names and contains the name of a protocol, the IANA assigned number for the protocol, and aliases for the protocol. Any protocol that `inetd` uses for a service must have an entry in the **/etc/protocols** file. It has one line per protocol in the following format:

```
protocol-name number [aliases]
```

Here, `protocol-name` is the name of the protocol such as `tcp`, number is the IANA assigned number for the protocol such as `6` for `tcp`, and aliases are alternative names for the protocols, such as TCP for `tcp`. Here is the line for the `tcp` protocol in the **/etc/protocols** file:

```
% grep ^tcp /etc/protocols
tcp    6    TCP  # transmission control protocol
%
```

20.10.1.4 Adding or Deleting inetd Services on PC-BSD

As an administrator, you can delete a service that `inetd` offers by simply removing the line for the relevant service from the **/etc/inetd.conf** file. Similarly, you can add a new service in `inetd` by adding a line for the corresponding service in the **/etc/inetd.conf** file.

After making the requisite changes in the **/etc/inetd.conf** file, you can reconfigure `inetd` dynamically so that it offers the updated set of services without having to terminate and restart it. You can do so by sending SIGHUP (no hang-up signal) to `inetd` using the `kill` or `killall` command. Table 20.25 shows four ways of performing this task. Note that the `killall` command is used with a process name and the `kill` with a process ID (PID). Note the use of *grave accents* (`) for the substitution of the `cat/var/run/inetd.pid` command, with the resultant execution of the command.

EXERCISE 20.27

Display the lines for all "internal" services listed in the **/etc/inet.conf** file on your system.

TABLE 20.25 Ways of Restarting inetd

1	`killall -HUP inetd`
2	`killall -1 inetd`
3	`kill -HUP 'cat /var/run/inetd.pid'`
4	`kill -1 'cat /var/run/inetd.pid'`

EXERCISE 20.28

Display the line in the **/etc/protocols** file for the UDP protocol. What numerical value is associated with the UDP protocol? How many protocols are defined in this file?

20.11 CONCURRENT CLIENTS

It seems obvious to think of concurrent servers to serve multiple clients simultaneously in an efficient manner, thereby reducing the average response time experienced by a client. However, the need for concurrent clients is not so obvious. There are several real-life examples of concurrent clients that are necessitated due to the application protocol. Many of the well-known services require concurrent clients, including SSH, TELNET, FTP, and HTTP. A client for each of these services has to deal with two descriptors: a standard input descriptor (keyboard) to read user input and a socket descriptor used by the client to communicate with the server process. Either descriptor may become ready for I/O asynchronously; a user may type the next command for the client using the keyboard any time that he/she desires to do so and the server response for a previous client request may arrive at the client-side socket at any instant of time, depending on the load on the server process and network traffic.

We explain the need for a concurrent client by using the example of a connection-oriented (TCP) SSH client. A TCP client for SSH expects input from two descriptors, one for the keyboard and the other for the socket that is connected to the SSH server-side socket. The client needs to receive user input (i.e., the next shell command to be executed on the remote host) entered through the keyboard and the server's response through the connected socket, as shown in Figure 20.32. Since the two descriptors become ready asynchronously, you may implement such concurrent clients by using the `select()` system call.

Another scenario under which a concurrent client becomes necessary arises because BSD UNIX does not allow independent processes to share main memory. The X Window System allows multiple clients to paint (i.e., redraw) a bitmapped display, so that the windows of the respective clients may be updated when required. Since the X display is

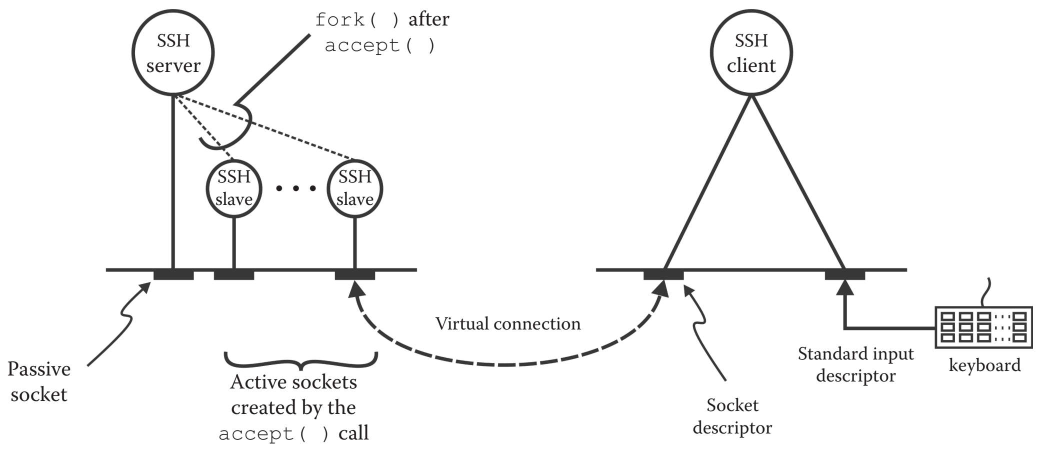


FIGURE 20.32 SSH client dealing with two descriptors.

memory mapped, the X server puts the information it receives from the various clients into one contiguous display buffer in the main memory. Depending on its location on the screen, each client window occupies a particular region of the buffer. The single-process server then uses the `select()` system call to handle asynchronous input from the client-side sockets and paint the screen accordingly.

You may also like to write the client for a particular service that simultaneously connects with multiple servers for a given service. Such concurrent clients are written using the master–slave model, where a slave process is created to handle a particular server. The domain names and/or IP addresses of the servers are passed to the client as command line arguments. One reason for writing such concurrent clients is to measure the response time for different servers.

EXERCISE 20.29

Write the names of five well-known Internet services that use concurrent clients. Explain why these services use concurrent clients.

20.12 WEB RESOURCES

Table 20.26 lists useful Web sites for UNIX IPC and related topics.

TABLE 20.26 Web Resources for the UNIX IPC and Related Topics

`http://www.iana.org/`	Webpage for Internet Assigned Numbers Authority (IANA): responsible for the global coordination of the DNS Root, IP addressing, transport protocol port numbers, and other Internet protocol resources
`http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml`	Service Name and Transport Protocol Port Number Registry
`http://tools.ietf.org/html/rfc6335`	RFC6335: Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry; forums, user groups, etc.
`http://www.iana.org/assignments/protocol-numbers`	Official names and numbers of the protocols
`https://www.freebsd.org/`	Home page for FreeBSD. Contains a lot of useful material, including FreeBSD source, manual pages, support, SVN repository, forums, user groups, etc.
`http://www.t1shopper.com/tools/port-number/N/` `http://www.adminsub.net/tcp-udp-port-finder/N`	Reports IANA assigned service on port 'N' (e.g., for N = 22 the page reports the service that is assigned port 22)
`http://www.tutorialspoint.com/`	Excellent site for tutorials on all kinds of programming languages, including UNIX system programming in C
`http://stackoverflow.com/`	Free question and answer site for rookie and professional programmers

SUMMARY

We discussed a number of important topics, primarily related to UNIX IPC, between related processes on the same machine, using related or unrelated processes on the same machine and related or unrelated processes on the same or different machines on a network. In doing so, we covered the use of pipes, named pipes (FIFOs), and sockets as communication channels and the related data structures, system calls, macros, and library functions. These calls and data structures deal with the creation of a communication channel, preparing it for communication where required, and using it for reading and writing messages between processes.

We discussed the issue of client–server design in detail, both for connectionless and connection-oriented communication between client and server processes. Our focus in this regard was the transport-level protocols, UDP and TCP. We described the need for different types of servers, including iterative and concurrent servers.

We explained the working of the various types of client–server models by designing and implementing application-level services similar to the well-known Internet services ECHO, TIME, and DAYTIME. We also discussed the design and implementation of a concurrent server using slave processes. Throughout the chapter, our discussions included algorithms for the client and server processes, call graphs, and interaction sequences. The call graphs describe the sequence of system calls for implementing client and server software individually, and interaction sequences describe the interaction between the client and server processes to implement the application protocol at hand.

We then covered the design of servers that offers multiple services using the `select()` system call. We built a simple concurrent server using the `select()` system call to implement the TIME and ECHO services for the UDP protocol. We also described how the UNIX superserver, `inetd`, works. Finally, we discussed the need for concurrent clients and how they work.

QUESTIONS AND PROBLEMS

1. What are little endian and big endian byte orders? Are these for data storage, data transmission, or both? What is network byte order? Give one example each of the processors that use these byte orders.

2. Show the contents of memory locations for the 32-bit hex number F9327CA5 using the little endian and big endian byte orders. Assume that the main memory is byte oriented.

3. What byte orders are used by the following CPUs: AMD64, Intel i7, Sun SPARC, Motorola 68000, IBM, PDP, and VAX? How did you obtain your answers?

4. What shell command can you use to display the name of the CPU used by your computer system? What byte order does this CPU use? How did you find out?

5. Write a C program that displays the size of the PPFDT and the largest descriptor value on your UNIX system. Show your program and a sample run of the program.

6. What happens to a reader process when it reads from a pipe that has no data in it? Explain the reason for your answer.
7. What happens to a writer process when it writes to a full pipe? Explain the reason for your answer.
8. What is the effect of using the following flags in the `pipe2()` system call?
   a. `O _ CLOEXEC`
   b. `O _ NONBLOCK`
9. How should the `pipe2()` system call be used so that it behaves like the `pipe()` system call?
10. What is the synchronization issue in the bounded-buffer reader–writer problem? Who handles synchronization when two UNIX processes communicate with each other using pipes: the reader process, the writer process, or someone else (specify)?
11. Write a program that demonstrates how the reader and writer processes behave while communicating through a widowed pipe.
12. Write a C program, `widowed.c`, that creates a widowed pipe and demonstrates that when a process reads from a pipe that cannot be written to, it results in the reader process receiving the eof message.
13. When you compile and run the `broken _ pipe.c` program in Section 20.4.2 on a Solaris machine, it terminates without generating the "`Broken pipe`" error message. How can you verify that the pipe was actually widowed and the program terminated when the `write()` system call was executed?
14. Use signal handling to generate the "`Broken pipe`" error message by the `broken _ pipe.c` program when you compile and run it on a Solaris machine. Write two versions of the program. The first version should terminate the program after displaying the "`Broken pipe`" message and the second version should continue after displaying the message and execute the statement after the `write()` system call that causes `SIGPIPE`. *Hint*: Use `signal()`.
15. Write a C program to demonstrate that a widowed pipe with its write end closed sends the eof message to the reader process. Show compilation and execution of the program.
16. Write a program that takes two command line arguments: the name of a text file that contains single-digit integer data and an integer to be searched from the sorted version of the data in the file. The program creates two children processes. The first child process reads the text file passed as the first argument, sorts the numbers (any sorting algorithm is allowed), and communicates the sorted numbers to the second child process via a pipe and exits. The second child process searches for the "majority number" in the sorted list of numbers, displays the majority number and sends the

sorted numbers to the parent process via another pipe, and exits. The parent process searches the sorted data that it receives from the child process and displays the number to be searched for, or a message in case the number is not found. All printing, input, reading (from files or the console) should be done by system calls. You cannot use any library functions.

17. Are FIFOs process persistent or file system persistent? Explain your answer.
18. When a pipe is created, two file descriptors are used in the PPFDT. How many descriptors are used when a FIFO is created? How many are used when a FIFO is opened?
19. What is the amount (size) of data that can be written into a FIFO atomically on BSD and Solaris? Where did you find answer to the question?
20. Compile and run the client–server model shown next. Make sure to run the server process first because it creates the FIFO used for communication between the client and server processes. What does the model do? Does the model work as expected? If not, what is wrong with it? Clearly identify the issues and their remedy.

```
% more fifo.h
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/errno.h>

extern int      errno;

#define FIFO   "/tmp/fifo"
#define PERMS   0666
#define SIZE    512

static char* message1 = "Hello, world!\n";
static char* message2 = "Hello, class!\n";
% more client.c
#include "fifo.h"

int main(void)
{
    char buff[SIZE];
    int fd;
    int n, size;

    /* Open FIFO. Assume that the server
       has already created them. */
    if ((fd = open(FIFO, 2)) == -1) {
        perror ("client open FIFO");
        exit (1);
    }
    /* client (fd); */
    size = strlen(message1);
    if (write(fd, message1, size) != size) {
        perror ("client write fd");
        exit (1);
    }
    if ((n = read(fd, buff, size)) == -1) {
        perror ("client read");
        exit (1);
    }
    else
        if (write(1, buff, n) != n) {
            perror ("client write stdout");
            exit (1);
        }
    close(fd);

    /* Remove FIFO now that we are done using it */
    if (unlink (FIFO) == -1) {
        perror("client unlink FIFO");
        exit (1);
    }
    exit (0);
}
% more server.c
#include "fifo.h"

int main(void)
{
    char buff[SIZE];
    int fd;
    int n, size;

    /* Create two FIFOs and open them for
       reading and writing */
    if ((mknod (FIFO, S_IFIFO | PERMS, 0) == -1)
                        && (errno != EEXIST)) {
        perror ("mknod FIFO");
        exit (1);
    }

    if ((fd = open(FIFO, 2)) == -1) {
        perror ("open FIFO");
        exit (1);
    }

    /* server (fd); */
    size = strlen(message1);
```

```
        if ((n = read(fd, buff, size)) == -1) {
            perror ("server read");
            exit (1);
        }
        if (write (1, buff, n) < n) {
            perror("server write stdout");
            exit (1);
        }
        size = strlen(message2);
        if (write (fd, message2, size) != size) {
            perror ("server write fd");
            exit (1);
        }
        close (fd);
    }
    %
```

21. Implement the client–server model given in Figure 20.9. The server offers the ECHO service. Test the model by running three concurrent clients through three terminal windows on your systems.
22. What system call would you use to create an IPv4 socket for stream-oriented communication that closes when the process executes a system call (or library call) in the `exec()` family?
23. What system call would you use to create an IPv6 socket for nonblocking stream-oriented communication that closes when the process executes a system call (or library call) in the `exec()` family?
24. What are the values of the following symbolic constants:
    a. `PF_INET`
    b. `PF_INET6`
    c. `SOCK_STREAM`
    d. `SOCK_NONBLOCK`
    e. `AF_UNIX`
    f. `PF_UNIX`
    g. `AF_INET`
25. How many domains of communication are supported by your system? How did you obtain your answer? If a symbolic constant is defined for this purpose, show this definition. *Hint*: Browse the **/usr/include/sys/socket.h** file.
26. Write the code snippet to create a socket for the stream-oriented style of communication using the TCP protocol under IPv4. Mark the socket "close-on-exec" and "nonblocking."

27. When is a socket half associated? When is it fully associated?
28. What feature of the TCP/IP protocol allows multiple distinct servers to run simultaneously on a host on the Internet and establish multiple concurrent communication sessions between the client and server programs running on the hosts on the Internet?
29. What type of socket-based communication requires that the sockets of the two communicating processes are fully associated?
30. What are the new versions of the `inet _ aton()` and `inet _ ntoa()` calls? Why would you prefer to use the new calls as opposed to the old calls?
31. Design and implement a client–server model for the ECHO service using the UNIX domain sockets.
32. What is the maximum length of the queue associated with the `listen()` system call on your system? What flavor of UNIX are you using? How did you obtain your answer?
33. Log on to a machine that runs at least one Internet server, such as SSH. Use the `netstat` command with appropriate options to determine the size of the queue associated with the passive socket of the server. Show the output of the command that you used for this purpose.
34. The `shutdown()` system call is used to close a socket. Why is it needed when you can use the `close()` system call to close a descriptor?
35. What type of network traffic is generated by the `connect()` system call for `SOCK _ STREAM` and `SOCK _ DGRAM` types of sockets under the `PF _ INET` and `PF _ LOCAL` domains?
36. What are the differences between the `accept()` and `accept4()` system calls?
37. Why must a process read data from a `SOCK _ STREAM` socket in a loop? Why is this not required in the case of a `SOCK _ DGRAM` socket?
38. What is the difference between active and passive sockets? How are they created? Why is the socket on which a server process waits for connection requests known as a passive socket?
39. Why are active sockets also known as ephemeral sockets?
40. Give two reasons each for the following system calls to fail:

    a. `socket()`

    b. `mkfifo()`

    c. `mkfifoat()`

   d. `bind()`
   e. `listen()`
   f. `connect()`
   g. `accept()`
   h. `select()`
   i. `shutdown()`

41. Suppose *S* is a connection-oriented concurrent server. If it is currently serving *K* clients, how many sockets and slave processes is the server using? What would these numbers be if *S* were a connectionless concurrent server? Explain your answers.
42. Write the code for the concurrent connectionless server for the ECHO service and test it with multiple clients accessing it simultaneously.
43. Write the code for the concurrent connection-oriented server for the ECHO service and test it with multiple clients accessing it simultaneously.
44. Design, code, and test the connection-oriented iterative client–server model discussed in Section 20.8.4.
45. What are the queue lengths associated with the passive sockets for all the servers running on your UNIX system? How did you obtain your answer? Show your work.
46. Some of the library functions for network programming, such as `inet _ ntop()`, have a size (or length) argument to specify the length in bytes of the destination address variable. Why is it needed?
47. What happens to the descriptor sets when the `select()` system call fails and returns `–1`?
48. What is returned by the `select()` system call when it returns due to the expiration of the timer?
49. When is it necessary to have concurrent clients? Give a few examples and give the structure of the code for an `HTTP` client.
50. Write the code for the single-process TCP ECHO server using the `select()` system call and test it using the ECHO client discussed in this chapter.
51. Write a thread-safe implementation of the `sleep()` system call that uses the `select()` system call. Explain why your implementation is thread safe.
52. Add the TCP DAYTIME service to `select _ server.c` in Section 20.9.
53. Modify `select _ server.c` so that it returns after 10.5 s instead of blocking until a descriptor becomes ready.

54. Why is there a need for concurrent clients? Write the concurrent client for the ECHO service that connects with multiple ECHO servers and displays the response time for each server. Show the source code for the client, its compilation, and a few sample runs.

55. What is the format of a line in the configuration file for `inetd`, the UNIX super-server? What is the meaning of each field?

56. What types of services are offered through the UNIX `inetd`? Explain your answer.

57. What would happen if a heavily loaded service were offered through `inetd`? Explain your answer. What should the system administrator do under such circumstances?

58. Write down the code snippet that makes `inetd` dynamically reconfigurable by using the `SIGHUP` signal. Show only signal-handling code and the structure of the code that actually reconfigures.

59. How many server and client processes run when `inetd` is serving two TIME clients and one client each for the ECHO, TELNET, and FTP services? How many sockets are used on the server side under this setting? Explain your answer.

60. The ``kill -HUP `cat /var/run/inetd.pid` `` command may be used to make `inetd` reinitialize and restart it dynamically on a BSD system. What is the equivalent of this command on the Solaris system?

61. What is the PID of `inetd` on your system? Write down the command that you used for this purpose.

62. What is a concurrent client? Why are concurrent clients needed? List three reasons. Also, list three well-known Internet services that require the use of concurrent clients.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

CHAPTER 21

System Programming IV

Practical Considerations

Objectives

- To explain the concept of restarting system calls
- To describe thread-safe system calls
- To discuss the concept of writing a program that starts running in the background and becomes a daemon
- To discuss setting signals and umask for a daemon
- To describe the concept of allowing only a single copy of a program to run
- To discuss saving a daemon's identity at a known location
- To explain the issue of detaching the terminal from a process
- To describe the issue of changing the working directory
- To explain the need to close inherited standard descriptors and open standard descriptors
- To describe the removal of completed child processes from the system
- To show and discuss the structure of a truly concurrent, connection-oriented production server
- To cover the system calls, library calls, commands, and primitives

  `chdir(), close(),fcntl(), flock(), fork(), free(), getpid(), goto, malloc(), open(), perror(), read(), setsid(), signal(), sprintf(), sleep(), strlen(), umask(), wait3(), write()`

21.1 INTRODUCTION

In this chapter, we discuss some important issues related to system programming in general and the design of server processes in particular. The issues are blocking I/O, restarting system calls that are interrupted during their execution due to the arrival of a signal, using thread-safe system calls, reentrant code, ignoring signals and setting the umask of server processes, running server processes as daemons, ensuring that only one copy of a program may run at a given time, closing inherited descriptors in a process, ignoring I/O in a daemon, redirecting errors to a file, and cleaning up the child processes that have terminated. We have already discussed some of these issues in detail in Chapters 18–20. Here, we discuss some specific issues, particularly related to the design of server software. We describe two methods of file locking and use them both in the final version of our example server to ensure that the server will port to both PC-BSD and Solaris machines. Throughout the chapter we use defensive coding.

21.2 RESTARTING SYSTEM CALLS

We should write code that has the ability to restart system calls that are interrupted during their execution. Such situations arise under different scenarios depending on the system call. However, in all cases the system call is in some kind of waiting state. The wait may be any one of the many types, including waiting for an I/O device, a communication channel, or a child process to terminate. The system calls involved may be `read()`, `readv()`, `write()`, `writev()`, `ioctl()`, `fcntl()`, `wait()`, `waitpid()`, `wait3()`, `wait4()`, `wait6()`, `select()`, and so on.

We discussed blocking I/O in detail in Chapters 19 and 20 while explaining the concept of a communication channel called the *UNIX pipe*, which is used for communication between related processes. A `read()` or `write()` call may be interrupted when you perform a *blocking I/O* on a pipe. In the case of a pipe, blocking input means that the `read()` system call blocks, as there is nothing to read because the pipe is empty. The `write()` system call blocks if the pipe is full. Another example of blocking read is when the `read()` system call reads input interactively from a keyboard. The `read()` call blocks until the user enters keyboard input. Similarly, the `select()` system call blocks while waiting for an I/O request on a descriptor. Finally, a process may block while waiting for a child process to terminate using any of the calls in the `wait()` class.

If a system call is in a blocking state, it may be interrupted while waiting for a particular event to happen (the arrival of data into a pipe, the flow of data out of a pipe, the termination of a child process, etc.). Most modern UNIX implementations restart interrupted system calls automatically. If you are not sure whether your code will run on such a system, you need to write code to explicitly handle the restarting of an interrupted system call. The following code snippet may be used for this purpose.

```
repeat:
      if ((nr = read(fd, buf, SIZE)) == -1)) {
         if(errno == EINTR) /* if interrupted system call */
            goto repeat;
```

```
        /* handle other errors */
    }
```

4.2BSD was the first UNIX system that supported automatic restart for interrupted system calls. I/O-related calls are interrupted by a signal when they operate on slow devices or a communication channel such as a pipe. However, the calls in the `wait()` class are always interrupted when they receive a signal. This is a problematic situation and system designers introduced a feature in 4.3BSD that allows a process to disable the automatic restarting of system calls on a per-signal basis.

21.3 THREAD-SAFE SYSTEM CALLS

In this section, we provide practical considerations that are an extension of the material presented in Chapter 19 on threads. We define, discuss, and give simple examples of two important concurrency topics: thread safety and reentrant functions.

Simply stated, a *thread-safe function* can be called simultaneously from multiple threads, even when the invocations use shared data, because all references to the shared data are serialized. In other words, each thread accesses shared data on a mutually exclusive basis after locking the data using synchronization primitives like *spinlocks* or *semaphores*.

A *reentrant function* can also be called simultaneously from multiple threads, but only if each invocation uses its own data. Therefore, a thread-safe function is always reentrant, but a reentrant function is not always thread safe. This means that with only one copy of a thread-safe or reentrant function in main memory, multiple threads or commands may execute it simultaneously. Most applications running on time-sharing systems are thread safe or reentrant, including compilers, word processors, and editors. All functions defined in the Single UNIX Specification (SUSv3) are guaranteed to be thread safe, with the exception of those listed in Table 21.1.

You should not use non-reentrant functions in signal handlers because a signal may occur while the control is in a non-reentrant function and another signal is received. Clearly, in such cases the program may produce a wrong result because of a race condition caused by the multiple simultaneous executions of a non-reentrant function. For example, the `malloc()` function maintains a linked list of the dynamically allocated areas. We should not use `malloc()` in a signal handler because it may cause problems due to race condition. Suppose we use `malloc()` in a signal handler, a signal occurs, and control goes to the signal handler; the `malloc()` function runs as part of the signal handler and it is in the middle of updating the list pointers when the second signal occurs. The linked list will then become corrupt. Similarly, functions that access global (static) variables are also prone to being non-reentrant and non–thread safe. No system calls and no library functions are guaranteed to be reentrant that (a) use `malloc()` or `free()`, (b) use global data structures, or (c) are part of the Standard I/O library. All of the system calls we have discussed in this book are reentrant.

Operating systems and thread libraries provide synchronization primitives for writing thread-safe and reentrant functions, including semaphores and *mutexes*. We do not cover these primitives in this book but you may read more about them in books or Internet

TABLE 21.1 Non-Thread-Safe Functions

`asctime()`	`fcvt()`	`getpwnam()`	`nl_langinfo()`
`basename()`	`ftw()`	`getpwuid()`	`ptsname()`
`catgets()`	`gcvt()`	`getservbyname()`	`putc_unlocked()`
`crypt()`	`getc_unlocked()`	`getservbyport()`	`putchar_unlocked()`
`ctime()`	`getchar_unlocked()`	`getservent()`	`putenv()`
`dbm_clearerr()`	`getdate()`	`getutxent()`	`pututxline()`
`dbm_close()`	`getenv()`	`getutxid()`	`rand()`
`dbm_delete()`	`getgrent()`	`getutxline()`	`readdir()`
`dbm_error()`	`getgrgid()`	`gmtime()`	`setenv()`
`dbm_fetch()`	`getgrnam()`	`hcreate()`	`setgrent()`
`dbm_firstkey()`	`gethostbyaddr()`	`hdestroy()`	`setkey()`
`dbm_nextkey()`	`gethostbyname()`	`hsearch()`	`setpwent()`
`dbm_open()`	`gethostent()`	`inet_ntoa()`	`setutxent()`
`dbm_store()`	`getlogin()`	`l64a()`	`strerror()`
`dirname()`	`getnetbyaddr()`	`lgamma()`	`strtok()`
`dlerror()`	`getnetbyname()`	`lgammaf()`	`ttyname()`
`drand48()`	`getnetent()`	`lgammal()`	`unsetenv()`
`ecvt()`	`getopt()`	`localeconv()`	`wcstombs()`
`encrypt()`	`getprotobyname()`	`localtime()`	`wctomb()`
`endgrent()`	`getprotobynumber()`	`lrand48()`	
`endpwent()`	`getprotoent()`	`mrand48()`	
`endutxent()`	`getpwent()`	`nftw()`	

sources on pthread programming and/or UNIX system programming that discuss these topics more comprehensively.

21.4 RUNNING PROCESSES IN BACKGROUND: DAEMONS

We discussed in detail the execution of background processes and daemons in Chapter 10. A system process that provides services to users or processes is known as a *daemon*. A common use of daemons is in server processes. Examples of some commonly known daemons are the http (Web) server (httpd), Secure Shell server (sshd), UNIX superserver (inetd), printer server (lpd), and page server (pageda).

In this section, we discuss how you can write code for a process that, when executed, automatically becomes a daemon. One of the characteristics of daemons is that they are disconnected from standard files. Creating a daemon is rather simple. The issue boils down to creating a child process, letting the parent process exit, and allowing the child process to continue and become a daemon. The **daemon.c** program in the following session illustrates the concept. After the parent process exits, the child process, now a daemon, displays its PID before entering an infinite loop to read client requests and respond to them.

```
% cat daemon.c
#include <unistd.h>
#define SIZE 32
```

```
int main(void)
{
        pid_t pid;
        char buf[SIZE];

        pid = fork();
        if (pid == -1) {
            perror("Fork failed");
            exit(1);
        }
        /* Parent process */
        if (pid > 0) {
            exit(0);
        }
        /* Child process: Continues and becomes the server */
        /* process running forever as a daemon             */
        (void) sprintf(buf, "Daemon PID: %d\n", getpid());
        (void) write(1, buf, strlen(buf));
        while (1) {
            /* Wait for a client request */
            /* Serve the client request  */
            sleep(10);     /* Dummy code */
        }
}
% gcc46 -w daemon.c -o daemon
% ./daemon
Daemon PID: 97272
% ps
  PID TT  STAT    TIME COMMAND
93220  1  Ss   0:00.69 -csh (csh)
97272  1  S    0:00.00 daemon
97273  1  R+   0:00.01 ps
% kill -9 97272
% ps
  PID TT  STAT    TIME COMMAND
93220  1  Ss   0:00.69 -csh (csh)
97273  1  R+   0:00.01 ps
%
```

EXERCISE 21.1

Provide the names of 10 UNIX daemons and their purposes.

EXERCISE 21.2

Compile and run the preceding **daemon.c** program to make sure it works on your system as expected.

21.5 IGNORING SIGNALS

We discussed the issue of signals and signal handling in detail in Chapter 20. In this section, we emphasize that when you write code for a server process, you must handle signals appropriately.

Sometimes, you need to write server processes that can initialize their data structures using a text-based configuration file, and you want such servers to be able to reconfigure themselves dynamically without stopping. You can do so by sending the process a particular type of signal and invoking a function that reconfigures the server. The UNIX superserver, inetd, is an example of a server process that reconfigures itself on `SIGHUP`. Thus, it contains a line of code similar to the following:

```
signal(SIGHUP, sig_hup);
```

where the `sig _ hup()` function reconfigures the internal data structures of inetd. The configuration file for inetd is **/etc/inetd.conf**.

21.6 CHANGING UMASK

It is important to set the umask in the server process so that all the files, including logs, created by the server process have the desired access privileges, regardless of the mode specified in an `open()` or `creat()` system call. You can do so by using the `umask()` system call, as follows:

```
(void) umask(027);
```

We discussed in detail the setting of the umask at the command line and how the permission bits of a newly created file are set in Chapter 5. In the following session, we show the code for **daemon.c** after including signal handling and setting the umask. The compilation and running of the program on a PC-BSD system shows that the program works correctly and the daemon does not terminate when we send it the `SIGHUP` and `SIGINT` signals using the `kill` commands. Eventually, we terminate `daemon` using the `kill -9 16624` command. The code works as expected on the Solaris machine too.

```
% cat daemon.c
#include <unistd.h>
#include <sys/signal.h>

#define SIZE 32

int main(void)
{
        pid_t pid;;
        char buf[SIZE];

        pid = fork();
        if (pid == -1) {
            perror("Fork failed");
            exit(1);
        }
        /* Parent process */
        if (pid > 0) {
            exit(0);
        }
        /* Child process: Continues and becomes the server */
        /* process running forever as a daemon             */

        /* Ignore signals */
        signal(SIGHUP, SIG_IGN);
        signal(SIGINT, SIG_IGN);

        /* Set umask */
        (void) umask(027);

        /* Display Daemon's PID */
        (void) sprintf(buf, "Daemon PID: %d\n", getpid());
        (void) write(1, buf, strlen(buf));

        /* Code for the server */
        while (1) {
            /* Wait for a client request */
            /* Serve the client request  */
            sleep(10);     /* Dummy code */
        }
}
% gcc46 -w daemon.c -o daemon
% ./daemon
Daemon PID: 16624
% kill -0 16624
% ps
  PID TT  STAT    TIME COMMAND
12930  2  Ss   0:00.55 -csh (csh)
16624  2  S    0:00.00 daemon
16639  2  R+   0:00.01 ps
% kill -2 16624
% ps
  PID TT  STAT    TIME COMMAND
12930  2  Ss   0:00.56 -csh (csh)
16624  2  S    0:00.00 daemon
16658  2  R+   0:00.01 ps
% kill -9 16624
% ps
  PID TT  STAT    TIME COMMAND
12930  2  Ss   0:00.58 -csh (csh)
16671  2  R+   0:00.01 ps
%
```

EXERCISE 21.3

Repeat the preceding shell session to make sure that the **daemon.c** program works correctly on both PC-BSD and Solaris systems.

EXERCISE 21.4

What services does inetd, the UNIX superserver, offer on your system? List all the connection-oriented and connectionless services that the configuration file for inetd contains.

21.7 RUNNING A SINGLE COPY OF A PROGRAM

In this section, we discuss how we can make sure that only a single copy of a process runs. Some daemons are designed so that only a single copy of the daemon runs. One of the reasons for this may be that the daemon needs exclusive access to an object such as a file or device. There may be other reasons—for example, if multiple instances of cron start running, each would run a scheduled operation. This would not only result in duplicate operations but possibly an error too. Similarly, if a server goes down and two system administrators restart it, then the end result would be unpredictable.

You can ensure that a single instance of a daemon runs by including a code snippet at the beginning of the server program that accesses a lock of some sort. The lock may be a *spin lock* (i.e., a *binary semaphore*), a lock for exclusive access of a record, or a lock for exclusive access of a file. The actual code for the service offered by the server is executed only after this lock has been acquired, ensuring that only one copy of the program executes. In UNIX, daemons are usually designed to use a lock file for this purpose. The lock files for UNIX daemons are located in the **/var/spool** directory in PC-BSD and in the **/var/run** directory in Solaris.

Files may be locked for mutually exclusive access by using the `flock()` system call. Here is a brief description of the `flock()` system call.

```
#include <sys/file.h>
int flock(int fd, int operation);
```

Success: `0`
Failure: –1 and kernel variable `errno` set to indicate the type of error

Here, `fd` is the descriptor of an open file and `operation` specifies the type of access requested. The possible values of the `operation` argument are shown in Table 21.2.

The `flock()` system call may fail for the reasons listed in Table 21.3.

TABLE 21.2 Possible Values of the `operation` Parameter and Their Effect on the File Referred to by the Descriptor fd

Value of `operation`	Effect on the File Referred to by `fd`
LOCK_SH	Lock the file for shared access
LOCK_EX	Lock the file for mutually exclusive access
LOCK_NB	Do not block while locking the file (i.e., nonblocking locking)
LOCK_UN	Unlock the file

TABLE 21.3 Reasons for `flock()` to Fail

Reason for Failure	Value of `errno`
The `LOCK_NB` option was specified and the file was already locked	`EWOULDBLOCK`
The `fd` argument is not a valid descriptor	`EBADF`
The `fd` argument does not refer to a file	`EINVAL`
The `fd` argument refers to an object that does not support file locking	`EOPNOTSUPP`
No locks are available	`ENOLCK`

You can use the following piece of code to lock a file for mutual exclusion. You open or create the `LOCKFILE` and lock it for exclusive access using the `flock()` system call. We use the nonblocking call so that if the given file is already locked, the program terminates. Since we do not have write permission to the **/var/spool** and **/var/run** directories, we place the lock file in the **/usr/home/sarwar/Servers** directory—that is, a user's home directory in PC-BSD.

```
#include <sys/file.h>
#include <sys/stat.h>
#define LOCKFILE "/usr/home/sarwar/Servers/testd.lock"
#define LOCKMODE (S_IRUSR|S_IWUSR|S_IWGRP|S_IROTH)
...
/* Open (or create) LOCKFILE */
lock = open(LOCKFILE, O_RDWR | O_CREAT, LOCKMODE);
if (lock < 0) {
    perror("open failed to create LOCKFILE");
    exit(1);
}
/* Lock LOCKFILE for mutually exclusive access */
if (flock(lock, LOCK_EX | LOCK_NB)) {
    perror("flock failed to obtain exclusive lock for file");
    exit(1);
}
```

The `flock()` system call is simple and portable across `fork()`. However, it is not available on all UNIX platforms, including Solaris. The `fcntl()` system call may be used for performing such operations on open files that may not be performed with other system calls, including locking files for shared or exclusive access. The `fcntl()` system call is a little harder to use— it does not hold locks across `fork()`, the lock on a file is released when a `close()` system call is used on its descriptor—but it works on most UNIX systems, including Solaris. Here is a brief description of the `fcntl()` system call.

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd, /* arg */ ...);
```

Success: `0`
Failure: –1 and kernel variable `errno` set to indicate the type of error

The `cmd` argument is the operation to be performed on the open file with descriptor `fd`. The data type, value, and use of the last argument depend on the value of the `cmd` argument. In this section, we are only interested in discussing commands that are used for file locking. There are two commands that may be used for this purpose: `F _ SETLK` and `F _ SETLKW`. Each command sets or clears a file segment lock according to the description of the lock given in the third argument via a pointer to a variable of type `struct flock`. The lock may be shared (read) or exclusive (write).

With `F _ SETLK` as the third argument, the `fcntl()` call returns immediately with a value of –1 if a lock cannot be set. The `F _ SETLKW` command is similar to the `F _ SETLK` command, except that in the case of `F _ SETLKW` the calling process waits (i.e., blocks) if other locks have blocked a shared or mutually exclusive lock. The process stays blocked until the request is satisfied. During this wait, if `fcntl()` is interrupted, it returns –1 and `errno` is set to `EINTR`.

The `flock` structure has at least the following fields:

```
struct flock {
   short   l_type;    /* lock operation type                   */
   short   l_whence;  /* lock base indicator                   */
   off_t   l_start;   /* starting offset from base             */
   off_t   l_len;     /* lock length in consecutive bytes;     */
                      /* l_len == 0 means until end of file    */
   int     l_sysid;   /* system ID running process holding lock */
   pid_t   l_pid;     /* process ID of process holding lock    */
   ...
}
```

The value of `l _ whence` may be `SEEK _ SET`, `SEEK _ CUR`, or `SEEK _ END`, to indicate that the relative offset `l _ start` bytes is from the start of the file, the current position of the file pointer, or the EOF, respectively. The file will be locked if `l _ len` and `l _ start` are set to 0 each and `l _ whence` is set to `SEEK _ SET`. The `l _ type` field is set to `F _ RDLCK` for shared (read) lock and `F _ WRLCK` for exclusive (write) lock.

In order for our code to be portable to Solaris, we have made a few changes to it. These changes are as follows. After these changes have been made, the code will run on both PC-BSD and Solaris systems.

1. Include the following header files in the already existing list of header files.

   ```
   #include <fcntl.h>
   #include <unistd.h>
   #include <sys/file.h>
   #include <sys/stat.h>
   ```

2. Create the ~/**Servers** directory on your system.

3. The preprocessor directive to define `LOCKFILE` on Solaris and PC-BSD systems should be as follows:

```
#ifdef __sun
#define LOCKFILE "/export/home/sarwar/Servers/testd.lock"
#else
#define LOCKFILE "/usr/home/sarwar/Servers/testd.lock"
#endif
```

4. Use a variable of the `struct flock` type.

```
struct flock lock;
```

5. The conditional compilation of file-locking code for Solaris and PC-BSD replaces the code for file locking on PC-BSD.

```
#ifdef __sun
        /* Set lock */
        lock.l_start = 0;
        lock.l_len = 0;
        lock.l_type = F_WRLCK;
        lock.l_whence = SEEK_SET;

        /* Lock LOCKFILE for mutually exclusive access */
        if (fcntl(lockfd, F_SETLK, &lock)) {
            perror("Daemon already running");
            close(lockfd);
            exit(1);
        }
#else
        /* Lock LOCKFILE for mutually exclusive access */
        if (flock(lock, LOCK_EX | LOCK_NB)) {
            perror("Daemon already running");
            close(lockfd);
            exit(1);
        }
#endif
```

We enhance the **daemon.c** program discussed in Section 21.6 with the code fragments discussed in this section, according to steps a) to e), for compilation and execution on both PC-BSD and Solaris machines. The following session shows the new version of the **daemon.c** program, its compilation, and the running of the executable code on a PC-BSD machine. The program works correctly on our Solaris machine too. Because of the lock, only one instance of the daemon may run at any given time.

```
% cat lock_daemon.c
#include <fcntl.h>
#include <unistd.h>
#include <sys/file.h>
#include <sys/signal.h>
#include <sys/stat.h>
```

```
#define SIZE 32
#define LOCKMODE (S_IRUSR|S_IWUSR|S_IWGRP|S_IROTH)
#ifdef __sun
#define LOCKFILE "/export/home/sarwar/Servers/testd.lock"
#else
#define LOCKFILE "/usr/home/sarwar/Servers/testd.lock"
#endif

int main(void)
{
        pid_t pid, lockfd;
        char buf[SIZE];
        struct flock lock;

        pid = fork();
        if (pid == -1) {
            perror("Fork failed");
            exit(1);
        }
        /* Parent process */
        if (pid > 0) {
            exit(0);
        }
        /* Child process: Continues and becomes the server */
        /* process running forever as a daemon              */

        /* Ignore signals */
        signal(SIGHUP, SIG_IGN);
        signal(SIGINT, SIG_IGN);

        /* Set umask */
        (void) umask(027);

        /* Allow only a single copy of the daemon */
        /* Open (or create) LOCKFILE              */
        lockfd = open(LOCKFILE, O_RDWR | O_CREAT, LOCKMODE);
        if (lockfd == -1) {
            perror("open failed to create LOCKFILE");
            exit(1);
        }
#ifdef __sun
        /* Set lock */
        lock.l_start = 0;
        lock.l_len = 0;
        lock.l_type = F_WRLCK;
        lock.l_whence = SEEK_SET;

        /* Lock LOCKFILE for mutually exclusive access */
        if (fcntl(lockfd, F_SETLK, &lock)) {
            perror("Daemon already running");
            close(lockfd);
            exit(1);
        }
#else
        /* Lock LOCKFILE for mutually exclusive access */
        if (flock(lockfd, LOCK_EX | LOCK_NB)) {
            perror("Daemon already running");
            close(lockfd);
            exit(1);
        }
#endif

        (void) sprintf(buf, "Daemon PID: %d\n", getpid());
        (void) write(1, buf, strlen(buf));
        while (1) {
            /* Wait for a client request */
            /* Serve the client request  */
            sleep(10);     /* Dummy code */
        }
}
% gcc46 -w lock_daemon.c -o daemon
% ./daemon
Daemon PID: 18521
% ps
  PID TT  STAT    TIME COMMAND
12930  2  Ss   0:00.62 -csh (csh)
18521  2  S    0:00.00 daemon
18528  2  R+   0:00.01 ps
% ./daemon
Daemon already running: Resource temporarily unavailable
% ps
  PID TT  STAT    TIME COMMAND
12930  2  Ss   0:00.63 -csh (csh)
18521  2  S    0:00.00 daemon
18555  2  R+   0:00.01 ps
% kill -9 18521
% ps
  PID TT  STAT    TIME COMMAND
12930  2  Ss   0:00.66 -csh (csh)
18587  2  R+   0:00.01 ps
%
```

EXERCISE 21.5

Repeat the preceding session on your system. Does it work?

21.8 LOCATING A DAEMON

In order to easily locate a daemon that is misbehaving or has crashed, system programmers normally record its PID in the lock file as soon as the daemon has been created and the relevant file has been locked for exclusive access. You may use the following code snippet after the file-locking code shown in Section 21.7 has been executed.

```
(void) sprintf(buf, "Daemon Name: %d\n", getpid());
(void) write(lock, buf, strlen(buf));
```

EXERCISE 21.6

Insert the preceding code in the **daemon.c** program used in Exercise 21.5. Compile and execute the program. Does it work? How do you know?

21.9 DETACHING THE TERMINAL FROM A DAEMON

Since daemons run in the background, there is no need to have any terminal attached to them because they are used to either provide operating system services to system administrators without human intervention (e.g., cron), provide services to the UNIX kernel (e.g., pageda), or communicate with other processes (clients) in a client–server model such as a Web server (httpd). You can detach the terminal from a daemon by using the `setsid()` system call. The `setsid()` system call creates a new session with the calling process as its session leader. The calling process is also the group leader of the newly created process group and has no controlling terminal. Here is a brief description of this call.

```
#include <unistd.h>
pid_t setsid();
```

Success: Process group ID of the new process group, which is the same as the PID of the caller process
Failure: –1 and kernel variable `errno` set to indicate the type of error

The call may fail and `errno` set to EPERM if the caller process is already a process group leader or the process group ID of another process matches the PID of the caller process.

You may use the following piece of code to detach the terminal from the process.

```
pid_t sid;
...
if ((sid = setsid()) == -1) {
     perror("setsid failed");
     exit(1);
}
```

EXERCISE 21.7

Enhance the **daemon.c** program created in Exercise 21.6 with the preceding code fragment. Compile and execute the program.

21.10 CHANGING THE CURRENT WORKING DIRECTORY

It is important to change the current working directory for a server to a known and safe directory, but not your home directory. Doing so is useful for several reasons:

1. It is easy to locate the `core` file if the system administrator terminates a misbehaving server or a server crashes (aborts) for some reason.
2. If a process is running in a directory, the file system that contains this directory cannot be unmounted without first terminating the service. Thus, you as system administrator would not be able to perform any tasks that require this file system (containing your home directory) to be unmounted.

It is difficult to identify a single directory that is appropriate for all servers. For this reason, the root directory is chosen for almost all servers. You make the root directory the current working directory for your server by adding the following piece of code to your server:

```
if ((chdir("/")) == -1) {
     perror("chdir failed");
     exit(1);
}
```

EXERCISE 21.8

Enhance the **daemon.c** program created in Exercise 21.7 with the preceding piece of code. Compile and execute the program.

21.11 CLOSING INHERITED STANDARD DESCRIPTORS AND OPENING STANDARD DESCRIPTORS

Since the child process, now the server process, inherits all open descriptors of its parent, it is important for the server process to close all of these open descriptors so that it does not fall short of descriptors while serving client requests. However, you first need to identify the open descriptors and then close them. If the child process does not inherit any open descriptor, then it needs to close only the standard descriptors. This is usually the case, unless you overlay the code for the child process with another executable using a call from the `exec()` family and the descriptors are not marked *close-on-exec*. You can close the standard descriptors using the following code snippet:

```
int fd;
...
for (fd = 2; fd >= 0; fd--)
if (close(fd) == -1) {
     perror("close failed");
     exit(1);
}
```

Most daemons do not explicitly deal with standard descriptors. However, many library functions assume that standard descriptors are open. Thus, in order to make such library calls work properly in your daemons, you should open the three standard files but attach them to a benign device. The UNIX black hole, **/dev/null**, is one such device that returns EOF on a read and consumes whatever you write to it. After closing all inherited descriptors you may use the following code fragment to open standard files and attach them all to **/dev/null**.

```
int fd;
...
if ((fd=open("/dev/null",O_RDWR)) == -1) {
     perror("open failed");
     exit(1);
}
(void) dup(fd);
(void) dup(fd);
```

EXERCISE 21.9

Enhance the **daemon.c** program created in Exercise 21.8 with the preceding code snippet. Compile and execute the program.

21.12 WAITING FOR ALL CHILD PROCESSES TO TERMINATE

As discussed in Chapter 20, several Internet services such as ftp are offered via server processes that provide services to client processes through multiple *slave processes*, one per client. The slave processes are precreated and/or created dynamically by the main server process when needed. The main server process is also known as the *master server* process. Such servers are known as concurrent, connection-oriented servers. Such a server runs in an infinite loop with the code structure shown in Figure 21.1.

Because such concurrent servers keep creating slave processes as client requests arrive, it is important to terminate a slave process properly and remove it from the system after it has provided its service to the client process. You can do so by using the `exit()` and `wait()`calls in tandem, in the slave and master processes, respectively. If the master server process does not remove the slave processes after they have provided their services, a large number of zombie processes will be created in the system. The problem is that since a concurrent server has to wait for the next client request after creating a slave process, it

```
while (1) {
    wait for a client request
    create a slave process when a client request arrives
    slave handles the client request
    dispose the slave process
}
```

FIGURE 21.1 Structure of a concurrent server.

```
void zombie _ gatherer(int);
...
int main(...)
{
   ...
   signal (SIGCHLD, zombie _ gatherer);
   while(1) {
      wait for a client request
      create a slave process when a client request arrives
      slave handles the client request
   }
   ...
}

void zombie _ gatherer(int signal)
{
   int status;

   while (wait3(&status, WNOHANG, 0) >= 0)
      ;
}
```

FIGURE 21.2 Structure of a concurrent, connection-oriented server.

cannot explicitly wait for a slave process to terminate by using the `wait()` system call. Fortunately, UNIX signals come to our rescue!

Recall that when a process terminates, the UNIX kernel sends the `SIGCHLD` signal to its parent. A concurrent server process may use this feature to intercept all `SIGCHLD` signals to remove the terminating slave processes from the system by using the code structure shown in Figure 21.2.

Recall that the `WNOHANG` option for the `wait3()` system call makes it a nonblocking call in the sense that if it does not find a child process that has performed `exit()`, it returns –1. When `wait3()` returns –1, the control returns from the `zombie _ gatherer()` function to the line of code in the main function that was interrupted by `SIGCHLD`.

A server like Apache will retain a number of child processes (slaves). The server always ensures a minimum number of children (e.g., ten), so that if you only have eight, it generates two more, and it also maintains a child until a number of uses has been reached. Also, a server can start a new child process as soon as an old one is killed off rather than waiting for a new request to come in. This is more efficient in that the request will not have to wait for a child to be spawned. In order to make it *fail safe*, Apache also uses three to five master processes, with one acting as *leader server* and the rest as *standby servers*. When the leader crashes or is brought down for maintenance, one of the standby servers automatically takes over as leader without disrupting the running services.

EXERCISE 21.10

Enhance the **daemon.c** program created in Exercise 21.9 with the code to remove the children that have completed their tasks and terminated. Compile and execute the program.

Since you are using the `wait3()` system call, make sure to include the **<sys/wait.h>** file in your program.

21.13 COMPLETE SAMPLE SERVER

In this section, we demonstrate the complete server program after including in the **daemon.c** program all of the features discussed in this chapter. Since we use the `wait3()` system call in the `zombie _ gatherer()` function, we need to include the **<sys/wait.h>** file in the list of existing header files. The complete server code is in the **test_server.c** file shown in the following session. It runs on both UNIX platforms, PC-BSD and Solaris.

```
% cat test_server.c
#include <fcntl.h>
#include <unistd.h>
#include <sys/file.h>
#include <sys/signal.h>
#include <sys/stat.h>
#include <sys/wait.h>

#define SIZE 32
#define LOCKMODE (S_IRUSR|S_IWUSR|S_IWGRP|S_IROTH)
#ifdef __sun
#define LOCKFILE "/export/home/sarwar/Servers/testd.lock"
#else
#define LOCKFILE "/usr/home/sarwar/Servers/testd.lock"
#endif

void zombie_gatherer(int);

int main(void)
{
        int fd, lockfd;
        pid_t pid, sid;
        char buf[SIZE];
        struct flock lock;

        pid = fork();
        if (pid == -1) {
            perror("fork failed");
            exit(1);
        }
        /* Parent process */
        if (pid > 0) {
            exit(0);
        }
        /* Child process: Continues and becomes the server */
        /* process running forever as a daemon            */
        /* Ignore signals */
        signal(SIGHUP, SIG_IGN);
        signal(SIGINT, SIG_IGN);

        /* Intercept SIGCHLD and cleanup the terminating child/
           Children */
        signal (SIGCHLD, zombie_gatherer);

        /* Set umask */
        (void) umask(027);

        /* Allow only a single copy of the daemon */
        /* Open (or create) LOCKFILE              */
        lockfd = open(LOCKFILE, O_RDWR | O_CREAT, LOCKMODE);
        if (lockfd == -1) {
            perror("open failed to create LOCKFILE");
            exit(1);
        }
#ifdef __sun
        /* Set lock */
        lock.l_start = 0;
        lock.l_len = 0;
        lock.l_type = F_WRLCK;
        lock.l_whence = SEEK_SET;

        /* Lock LOCKFILE for mutually exclusive access */
        if (fcntl(lockfd, F_SETLK, &lock)) {
            perror("Daemon already running");
            close(lockfd);
            exit(1);
        }
#else
        /* Lock LOCKFILE for mutually exclusive access */
        if (flock(lockfd, LOCK_EX | LOCK_NB)) {
            perror("Daemon already running");
            close(lockfd);
            exit(1);
        }
#endif

        /* Save daemon PID in the lock file */
        (void) sprintf(buf, "testd PID: %d\n", getpid());
        (void) write(lockfd, buf, strlen(buf));

        /* Detach terminal from the daemon
        if ((sid = setsid()) == -1) {
            perror("setsid failed");
            exit(1);
        }
        /* Change working directory */
        if ((chdir("/")) == -1) {
            perror("chdir failed");
            exit(1);
        }

        /* Close inherited standard descriptors */
        for (fd = 2; fd >= 0; fd--)
        if (close(fd) == -1) {
            perror("close failed");
            exit(1);
        }

        /* Open standard descriptors */
        if ((fd=open("/dev/null",O_RDWR)) == -1) {
            perror("open failed");
            exit(1);
        }
        (void) dup(fd);
        (void) dup(fd);

        /* Main server loop */
        while (1) {
            /* Wait for a client request */
            /* Serve the client request  */
            sleep(10);     /* Dummy code */
        }
}

void zombie_gatherer(int signal)
{
        int status;

        while (wait3(&status, WNOHANG, 0) >= 0)
                ;
}
%
```

In the following session, we compile and run this program on a PC-BSD machine to show how the daemon works as coded. The executable code is saved in the **testd** file. We run this daemon with the `testd` command. The output of the `ps` command shows that the PID of the daemon is 53163. The outputs of the `ls -l /usr/home/sarwar/Servers` and `more /usr/home/sarwar/Servers/testd.lock` commands show that the **testd.lock** file is in fact created and contains the PID of the daemon. When we try to rerun the daemon, we get an error message that the daemon is already running, confirming that, as programmed, only one instance of the daemon may run at any given time. We remove the daemon from the system with the `kill -9 53163` command. If you don't remove the daemon, it will continue to run even after you have logged out because it ignores the `SIGHUP` signal.

```
% gcc46 -w test_server.c -o testd
% ./testd
% ps
  PID TT  STAT    TIME COMMAND
51375  1  Ss   0:00.37 -csh (csh)
53163  1  S    0:00.00 testd
53182  1  R+   0:00.01 ps
% more ~/Servers/testd.lock
testd PID: 53163
% ls -l ~/Servers
total 5
-rw-------  1 sarwar  faculty  17 Apr 14 00:15 testd.lock
% ./testd
Daemon already running: Resource temporarily unavailable
% ps
  PID TT  STAT    TIME COMMAND
51375  1  Ss   0:00.39 -csh (csh)
53163  1  S    0:00.00 testd
53242  1  R+   0:00.01 ps
% kill -9 53163
%
```

Here is a test of the testd daemon on Solaris. Note that the daemon works as expected on the Solaris system too.

```
$ gcc -w test_server.c -o testd
$ ./testd
$ ps
  PID TTY         TIME CMD
 1661 pts/2       0:00 testd
 1662 pts/2       0:00 ps
 1528 pts/2       0:00 bash
$ more ~/Servers/testd.lock
testd PID: 1661
$ ls -l ~/Servers
total 9
-rw-------   1 sarwar   faculty       16 Apr 14 05:03 testd.lock
$ ./testd
Daemon already running: Resource temporarily unavailable
$ ps
  PID TTY         TIME CMD
 1661 pts/2       0:00 testd
 1667 pts/2       0:00 ps
 1528 pts/2       0:00 bash
$ kill -9 1661
$
```

EXERCISE 21.11

Compile and execute the **test_server.c** program discussed in this section. Does the program work on your system as expected? Report any error that your program generates and give reasons for the errors.

21.14 STRUCTURE OF A PRODUCTION SERVER

A production server must handle all of the issues discussed in this chapter. The structure of a concurrent, connection-oriented production server with true process-level concurrency is shown in Figure 21.3.

21.15 WEB RESOURCES

Table 21.4 lists useful Web sites for UNIX system programming and related topics.

SUMMARY

We discussed a number of important topics, primarily related to server design. We discussed the issue of interrupted system calls, the system calls that may be interrupted and the circumstances under which these interruptions may occur, and restarting interrupted system calls. We then discussed thread-safe and reentrant functions, their importance, and what may cause a function to be non–thread safe and non-reentrant.

1. Create a child process
2. Terminate the parent process, making the child execute in the background as a daemon
3. Set signals according to the requirements of the server (ignored, default action taken, or programmer defined action taken)
4. Set the umask
5. Open a file for mutually exclusive access to ensure that only one copy of the daemon may run at a given time; such a file is known as the lock file
6. Save the PID of the daemon in the lock file for quick identification of the daemon/server
7. Detach the terminal from the daemon
8. Change current working directory
9. Close inherited standard descriptors
10. Open the standard files and attach them to a benign device such as /dev/null
11. Execute the true server code using an infinite loop as shown below
    while (1) {   /* Master server process */
    a. Wait for a client request
    b. Accept the client request
    c. Create a slave process, hand over the client request to the slave process, and go back to wait for another client request
    d. Slave process serves the client request(s) and exits
    e. Remove from the system the slave process that has completed its wok
    }

FIGURE 21.3 Structure of a truly concurrent, connection-oriented production server.

TABLE 21.4 Web Resources for Practical Consideration in Server Design

`https://www.freebsd.org/`	Home page for FreeBSD. Contains a lot of useful material, including FreeBSD source, manual pages, support, SVN repository, forums, user groups, etc.
`https://www.freebsd.org/doc/handbook/`	The page for FreeBSD documentation. An excellent resource for anything you want to know about FreeBSD
`https://www.netbsd.org/docs/guide/en/chap-inetd.html`	An excellent page on the UNIX superserver, inetd
`https://www.freebsd.org/doc/handbook/network-inetd.html`	Another excellent page on the UNIX superserver, inetd

We then discussed the various issues important for the design of a production server. These issues are: creating a daemon, setting signals and umask, ensuring that only a single copy of a process may run at a given time, saving the PID of the daemon at a known place, detaching the terminal from the daemon, changing the working directory, closing inherited standard descriptors, and opening standard descriptors and attaching them to a benign device; finally, we presented a simple server that has all of these features. Along the way, we discussed two methods of file locking using the `flock()` and `fcntl()` system calls. At the end, we outlined the steps necessary for the creation of a concurrent, connection-oriented production server that uses slave processes to serve clients.

Throughout the chapter, we showed the uses of the various system calls and library functions in small C code fragments and programs to illustrate different issues related to the design of UNIX servers. Finally, we combined all the code fragments to get the final version of a server that runs on both Solaris and PC-BSD systems. In order to make the code portable across all UNIX systems, remove the conditional compilation and only compile the file-locking code based on `fcntl()`.

QUESTIONS AND PROBLEMS

1. When does the issue of restarting a system call arise?
2. What is a thread-safe system call? What are the issues that make a system call not thread safe.
3. What is reentrant code? Why is this property important in a program? Give examples of a few common applications or tools that are reentrant.
4. What are the implications of reentrant code from the point of view of memory management on a time-sharing system? Give a small example to illustrate your answer.
5. Which of the following functions are thread safe and/or reentrant? Explain your answers.

a.
```
int test;
void swap(int *first, int *b)
{
    test = *first;
    *first = *b;
    *b = test;
  }
```

b.
```
int test;
void swap(int *first, int *b)
{
    int q;
    q = test;
    test = *first;
    *first = *b;
    *b = test;
    test = q;
}
```

c.
```
int sec_var = 1;
int first()
{
    sec_var = sec_var + 2;
    return sec_var;
}
```

d.
```
int sec()
{
    return first() + 2;
}
```

e.
```
int first(int i)
{
    return i + 2;
}
```

f.
```
int sec(int i)
{
    return first(i) + 2;
}
```

6. What is a daemon? Name five daemons in a typical UNIX system. Write down the purpose of each.
7. Why should a server close all inherited descriptors?
8. Why does a server process open the standard descriptors after closing all inherited descriptors? Why are the newly opened standard descriptors attached to a device like **/dev/null**?
9. What kind of locks can be placed on a file using the `flock()` system call? What is the purpose of each lock?
10. Suppose a process has locked a file **ABC** for exclusive access using the `flock()` system call. The process forks a child and the child unlocks the file. Will the parent still have exclusive access to file? Explain your answer.
11. Why should a daemon be coded such that only one instance of it may execute at any given time? What is the most common method of doing so? Write a small server program in C that uses this method. Compile and run the program, and show that your method works.
12. Consider the code shown in Section 21.7 for locking a file for mutually exclusive access by a process. Change

```
#define LOCKFILE "/usr/home/sarwar/Servers/testd.lock"
```

to

```
#define LOCKFILE "~/Servers/testd.lock"
```

Make a program out of the code with this change. Compile and run the program. Does the program work as expected? If it produces any errors, identify the buggy statement in the program, correct it, and show that the corrected version works properly.

13. The `flock()` and `fcntl()` system calls may be used to lock files for mutually exclusive access. Which of these locks may be passed on to children? Which are supported on most UNIX platforms?
14. Write a connection-triggered server that becomes a daemon and provides the date service; further, it allows only a single copy of the daemon to run. Test run the server with a client and show the complete session that captures the running of the server and client. Test the client–server model by running it on the same machine and on different machines.
15. In some UNIX systems, including Solaris, the signal handler is established again after the signal has occurred. Why? On such systems, what would happen if:
    a. The signal handler is not reestablished after the signal has occurred?
    b. The signal occurs before the signal handler is reestablished?
16. The `malloc()` and `free()` functions are not reentrant. What do you think are the reasons?
17. What is the purpose of the `WNOHANG` option in the `wait3()` system call?

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

CHAPTER 22

UNIX X Window System GUI Basics

Objectives

- To explain the relationship of the components of an X Window System–based graphical user interface to UNIX
- To describe the basic concepts and implementation of the X Window System
- To give an overview of the PC-BSD FVWM window management system
- To describe and give examples of client application program coding for the X Window System
- To cover the commands and primitives

  `xterm`

22.1 INTRODUCTION

This chapter presents the major objectives as follows. To get the most out of it, a beginner should go through each topic listed in the order shown.

- *X Window System model*: We first define the X Window System, a *network protocol* for graphical interaction between a user and one or more computer systems running UNIX. This means that it is a software system specifically designed to work over a network to pass user-generated events to an application program, and then channel graphical responses as graphical output back to the user. The forms of interactivity, via event-driven input and multiwindow display output, are detailed from the user's perspective. We illustrate and explain the basic X Window System at a high level —that is, closer to the user rather than the hardware in terms of operability and functionality.

- *Desktop management systems*: Since the chief arbiter of the user–computer interactive dialog is the *desktop management system* or *window management system*, we then define and detail the functionality of these kinds of programs. We give a functional overview of a generic window manager, named FVWM—most often used in what is termed a *nonintegrated* installation—to expose you to the look and feel of a "vanilla" window manager and its capabilities. We describe the operations of the KDE4 desktop management systems used in PC-BSD, the most important implementations of a desktop manager in our base UNIX system. We describe and give an overview of the default window manager in this desktop management system.

- *X Window System client application program coding*: We then go on to describe how to work with the X Window System at a lower level of operability by giving a basic description of how to write a client application program for the X Window System, and then by showing the use of some programming toolkits that facilitate this process. We give basic examples of various methodologies: programming using Xlib/XCB and programming using Qt toolkit libraries for the KDE4 desktop manager in PC-BSD.

22.1.1 User–Application Software Interaction Model

When you sit at a computer and work with an application program to accomplish specific tasks, you are primarily concerned with achieving the results that the computer provides. You are shielded from the details of exactly how the computer turns the motions of your hands and fingers into those results. One way of seeing the process that the computer goes through is shown in Figure 22.1, where you, the *user*, harness the intermediary facilities of software components, either locally on the same workstation, or globally over a network or the Internet, to work with the application program.

The fundamental assumption of this chapter is that a graphical user interface (GUI) can be used to most efficiently control the dialog between a single user and an application program running on a stand-alone or networked computer, using the intermediaries of the X Window System and the UNIX operating system. The components of a user's dialog

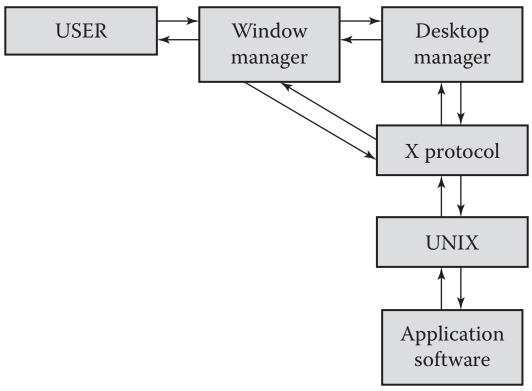


FIGURE 22.1 General components of a GUI.

with an application program can be simplified to the software component blocks shown in Figure 22.1.

For example, a user presses a mouse button to signify a graphical "pick" in an application window shown on screen. That choice, or event, is recognized by and acted upon by the window manager controlling that window. This event is passed along to the desktop manager, which uses the protocols of the X Window System to pass the request to UNIX. UNIX then passes the request to the application software program for further disposition. Another example is the reverse of the previous one. An application software program generates a request for graphical service, passes this request to UNIX, which in turn passes the request via the X Window System protocols to the desktop manager and window manager to display the graphical request on the screen of the user's computer.

It is important to note that you can either use the facilities of a *nonintegrated GUI system* (i.e., one that only uses the functions of a window manager and the X Window System to work within the UNIX environment) or an *integrated GUI system*, which uses the desktop manager (and perhaps a session manager) as an intermediate software link in the chain shown in Figure 22.1. For example, if you install UNIX on your computer, you may wish to only install the X Window System and a default window manager, such as FVWM. This would constitute a nonintegrated GUI installation. The event generation chain of interactivity as seen in Figure 22.1 would start at the user and proceed through the window manager and then jump to the X Window System. An integrated approach would involve installation of a desktop manager such as KDE4. The event generation chain of interactivity as seen in Figure 22.1 would start at the user and proceed through the window manager, then desktop manager, and then to the X Window System. Most users will use the integrated installation. Section 22.2 deals exclusively with a nonintegrated installation. Section 22.3 deals with an integrated installation.

If you have done a nonintegrated installation of the X Window System and UNIX—for example, by installing TrueOS without a GUI and then built the X Window System from source code or package—and have not specified that the X Window System start automatically when your computer boots up, you must type `startx` at the UNIX command prompt in order for the X Window System and a default window manager to "take control" of your display.

We have done an integrated installation of UNIX, with PC-BSD and the KDE desktop environment. Be advised that when we show a nonintegrated installation example, we are really showing a "mock" installation example, because we started from an integrated environment to begin with!

We do not show the installation of an operating system like TrueOS (which, by default, has a CUI exclusively) with the X Window System and FVWM built on top of it.

22.2 BASICS OF THE X WINDOW SYSTEM

22.2.1 What is the X Window System Similar to and What Advantage(s) Does it Have?

Contemporary user–computer interactivity falls into two basic categories, as mentioned in previous chapters. In one category, where a *character user interface* (CUI) is implemented, the user types commands on a command line using a keyboard, and components of the operating system handle this input and take appropriate action. In the other category, the user gives input via a *graphical user interface* (GUI), and components of the operating system take appropriate action. Of course there are also hybrid styles of interactivity which are a mixture of these two categories. Up to this point in the book, you have relied almost entirely on a CUI to activate the functionality of UNIX. In this chapter, you will be introduced to a UNIX GUI system, known as the X Window System. The two foremost questions for the beginner concerning the X Window System are: What is it similar to, and what advantage does it give me over the traditional UNIX CUI?

The answer to the first question is twofold. The X Window System is a network protocol developed to provide a GUI to the UNIX operating system; on the surface it appears to the user like other popular operating system window managers, such as those found on an Apple or in Microsoft Windows. (The current version of the X Window System (Release 7.7) is what we used for our base PC-BSD UNIX system. There is an important differentiation to make here between *window system*, *window manager*, and *desktop manager*. Briefly stated, a window system provides the generic functionality of the GUI, a window manager simply has particular implementations of the functionality provided by the window system, and a desktop manager provides a graphical method of interacting with the operating system. For example, interactive resizing of a window by the user is a generic function of a window system, whereas using icons or slider buttons is how it is accomplished in a particular window manager. The desktop manager provides the user with the graphic means to work with operating system functions such as file maintenance. A desktop manager might present a picture of folders connected in a tree-like structure and allow the user to manipulate files in those folders by dragging and dropping icons. Certainly, a modern window manager can include some or all of the functional features of a desktop manager. The role that a window manager plays in the X Window System and examples of window manager functionality are given in Section 22.2.3.

EXERCISE 22.1

What is the name of the desktop management system in PC-BSD?

EXERCISE 22.2

What is the name of the desktop management system in Solaris?

The first question can also be answered by giving an analogy: what the X Window System does for a user of networked computers is exactly like what an operating system does for the user of a stand-alone computer. On a stand-alone computer, the complex details of managing the resources of the hardware of the computer to accomplish tasks are left to the operating system. The user is shielded by the operating system from the complex hardware details of actually accomplishing a task, such as copying a file from one place to another on a fixed disk. On a system of networked computers, the X Window System manages the resources of the hardware of possibly many computers across the network to accomplish tasks for an individual user. Also, in a networked, distributed-system environment, where many machines are hooked up via a communications link, the X Window System serves *transparently* as a manager of the components of your interaction with application programs and system resources; in other words, you can run an application program on a machine that you are not sitting in front of, and the mechanics of interaction with the application work exactly as if the application were executing on a stand-alone machine that was right on the desk in front of you.

The most obvious answer to the second question is that you are able to quickly and easily accomplish predefined tasks by using a GUI under UNIX. For example, dragging icons to delete files is faster than typing commands to do the same thing, particularly if the file names are long and complex! Another not so obvious answer is that your style of interaction with the operating system will be very similar to your style of interaction with applications. For example, modern computer programming and engineering applications are graphics based, and have a common look and feel; pull-down menus almost always include functions such as cut, copy, paste, and so on. Having a GUI for UNIX makes for uniformity of interaction between operating system and application.

22.2.2 The Key Components of Interactivity: Events and Requests

When you work with a computer, you provide input in a variety of ways, and the computer, after doing some processing, gives you feedback in return. Limiting this feedback to text and graphics, usually the computer responds by displaying information on the screen.

On a modern computer workstation, you are able to use several devices, such as keyboard, mouse buttons, digitizing tablet, trackball, and so on, to provide input to an application program in a style of interaction known as *interrupt-driven interaction*. The application is processing data or in a wait state until signaled by a particular input device. Interrupts are known as input *events* from one or more devices, which can be ordered in time by forming a list or *queue*. With applications written for the X Window System, the client application can then process this queue of input events, do the work necessary to form responses to the events, and then output the responses as *requests* for graphical output to the server. A schematic illustration of this is shown in Figure 22.2.

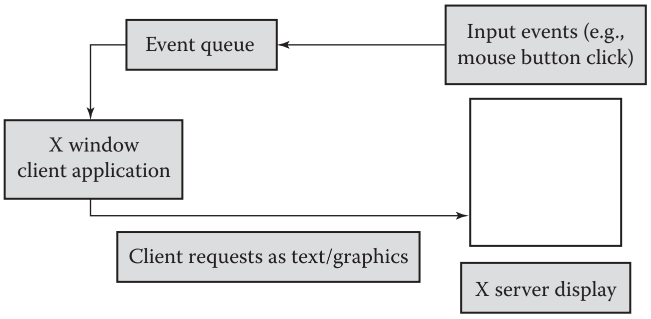

Event queue
Input events (e.g., mouse button click)
X window client application
Client requests as text/graphics
X server display

FIGURE 22.2 Event–request model.

A key concept of the X Window System that sometimes causes confusion is the difference between *server* and *client*. One possible cause for confusion here is that traditionally, on a computer network, a server is thought of as a machine that serves files to many other machines, which is certainly a different function than an X Window System server. In the X Window System, a server is the hardware and/or software that actually takes input from and displays output to the user. For example, the keyboard, mouse, and display screen in front of the user are part of the server; they graphically *serve* information to the user. The client is an application program that connects to, receives input events from, and makes output requests to the server. Be aware that sometimes in X Window System jargon, the client is spoken of as a hardware device, like a workstation or computer. We will always use the term *client* to refer to application program code, rather than to a piece of hardware. In the X Window System, a server and client can exist on the same workstation or computer, and use interprocess communication (IPC) mechanisms, such as UNIX pipes or sockets, to transfer information between them. A *local client* is an application that is running on the same machine that you are sitting in front of. A *remote client* is an application that is running on a machine connected to your server via a network connection. *Whether a client application is local or remote, it still looks and feels exactly the same to the user of the X Window System.*

Looking at Figure 22.3, you will see three client applications, X, Y, and Z, displaying their output on an X Window System server. Each of these applications is running on a different machine. Client X is running on a machine linked to the X Window System server via a LAN hookup, an Ethernet. Client Y is running on a machine linked to the X Window System server via a wide area network, the Internet. Client Z is running directly on the workstation that is the server, and uses UNIX sockets to display output requests on the server screen. Something not illustrated in Figure 22.3 is that each of the clients X, Y, and Z gets input events via this server as well.

Another critical aspect of the X Window System is that the GUI for each client is independent of the GUI of the window manager itself. In other words, each client application can open a window on the server screen, use its own style of GUI buttons, icons, pull-down

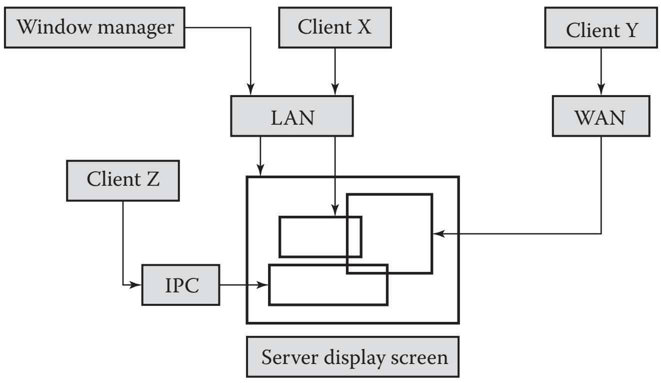


FIGURE 22.3 Client and server topologies.

menus, and so on, and the window manager, which is simply just another client application, handles the display of all other client windows. Figure 22.3 illustrates this point.

EXERCISE 22.3

If the client can queue events, do you think it would be advantageous for the server to queue requests? Why?

EXERCISE 22.4

From what you know of network programming in UNIX, is the meaning of client–server the same in network programming as it is in the X Window System? If it is not the same, what is the salient difference?

The important aspect of the window manager being just another client of the X Window System server is that you can use any of the available X Window System window managers to suit your particular needs. You can even use your own window manager, if you have the time and resources to write the program code for one! It is worth noting that only one window manager can be active on a given server at one time.

22.2.3 The Role of a Window Manager in the User Interface, and FVWM for PC-BSD

FVWM, the X Window System window manager, is not installed by default on our PC-BSD system. So, in order to follow along with the following sections, you need to use the AppCafe to install it. On a PC-BSD system, if that has not already been done by the system administrator on your system you can do this for yourself by using the procedures of the AppCafe.

Once you have installed the FVWM window manager, you can log out and then log in again to your PC-BSD system, and the initial login window that appears will allow you to choose FVWM as the window manager.

The desktop management system that we install by default under our base PC-BSD UNIX system is KDE4, and the window manager that we use to show a nonintegrated environment look and feel is release 2.6.5_3 of FVWM.

As implied in the previous section, the user interface of the X Window System has two basic parts: the *application user interface*, which is how each client application presents itself in one or more windows on the server screen display, and the window manager or *management interface*, which controls the display of and organizes all client windows. The application user interface is built into (i.e., written in a high-level programming language like C or C++ along with) a client application, and utilizes subroutine calls to a library of basic X Window protocol operatives. In this section we concentrate on and discuss the general functions that a window manager provides to control the appearance and operability of client application windows; in particular, we examine FVWM, a standard window manager that comes ready to run with the X Window System. All window managers in the X Window System are highly customizable, both by the system administrator, and by the user. Compare this to other window managers that are built into the operating system and cannot be customized to any real extent. The appearance and functionality of FVWM can be modified by you, as we will show in the final subsections of this section.

22.2.3.1 Functions and Appearance of the Window Manager Interface

Similar to the look and feel and functionality of Microsoft Windows or OS X, you will recognize many of the general functions that a UNIX X Window System window manager provides, shown in Table 22.1. These functions are particular implementations of possibly more than one generic window system service, those provided by the X Window System protocol.

22.2.3.2 The Appearance and Operation of FVWM

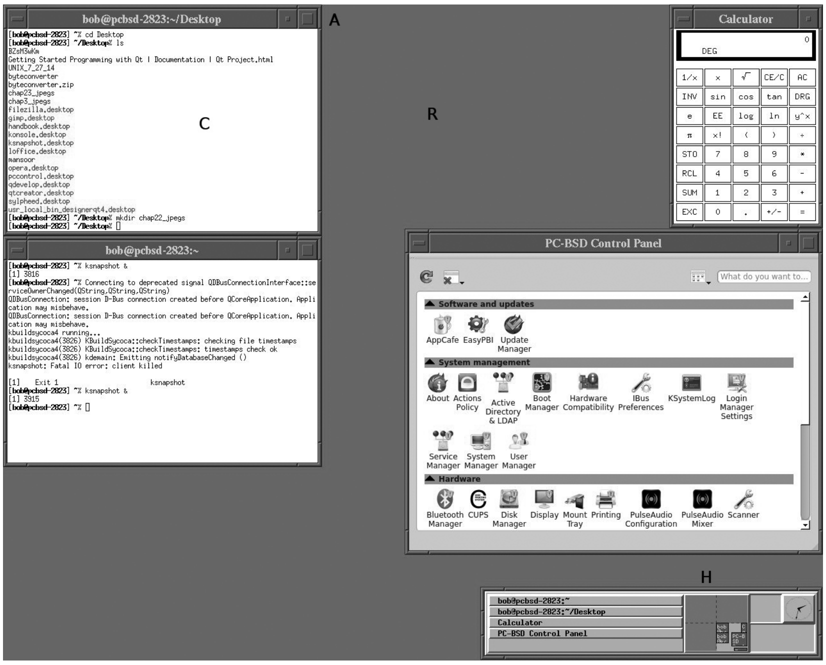
It is worth examining and identifying the components of a typical X Window System window display. Figure 22.4 shows a full-screen display using the FVWM window manager, and illustrates some particular examples of the implementation of the functions found in Table 22.1.

There are a few general things to notice about the screen display shown in Figure 22.4. The background of this screen display is known as the *root window*, labeled "R." All other windows that open on the screen are *children* of this *parent* window. In fact, a single parent window of one client can itself spawn many subwindows, which are all children of that client's parent. An interesting and important aspect of this relationship is found when parent windows obscure or cover child windows, or when child windows cannot exist outside of the frame defined for the parent window. In the first instance, simply uncovering the child, if this is possible, allows you to operate in the child window. In the second instance, the parent window may become very cluttered due to the existence of too many uncovered children on top of it. Figure 22.4 shows no covered windows, and visually is similar to what is known as a *tiled* display.

When you hold down the left-most (first), middle, or right-most (third) mouse button when the graphics cursor is in the root window, you are given the opportunity to utilize

TABLE 22.1 Window Manager General Functions

Item	Function	Description
A	(De)iconify window	Reduce window to a small, representative picture, or enlarge to a full size window
B	Create new window	Launch or run a new client application
C	CUI to operating system	Allows user to open one or more windows and type commands into those windows
D	Desktop management	Graphical file maintenance, speed buttons, special clients like time-of-day clock
E	Destroy window	Close connection between server and client
F	Event focus	Specifies which client is receiving events from devices like mouse, keyboard, etc.
G	Modify window	Resize, move, stack, tile one or more windows
H	Virtual screens	More than one screen area mapped onto the physical screen of the server
I	Pop-up/pull-down menus	Utility menus activated by holding down mouse buttons to run client applications



```
[bob@pcbsd-2823] ~% cd Desktop
[bob@pcbsd-2823] ~/Desktop% ls
BZsM3wKm
Getting Started Programming with Qt | Documentation | Qt Project.html
UNIX_7_27_14
byteconverter
byteconverter.zip
chap23_jpegs
chap3_jpegs
filezilla.desktop
gimp.desktop
handbook.desktop
konsole.desktop
ksnapshot.desktop
loffice.desktop
mansoor
opera.desktop
pccontrol.desktop
qdevelop.desktop
qtcreator.desktop
sylpheed.desktop
usr_local_bin_designerqt4.desktop
[bob@pcbsd-2823] ~/Desktop% mkdir chap22_jpegs
[bob@pcbsd-2823] ~/Desktop% 
```

```
[bob@pcbsd-2823] ~% ksnapshot &
[1] 3816
[bob@pcbsd-2823] ~% Connecting to deprecated signal QDBusConnectionInterface::se
rviceOwnerChanged(QString,QString,QString)
QDBusConnection: session D-Bus connection created before QCoreApplication. Appli
cation may misbehave.
QDBusConnection: session D-Bus connection created before QCoreApplication. Appli
cation may misbehave.
kbuildsycoca4 running...
kbuildsycoca4(3826) KBuildSycoca::checkTimestamps: checking file timestamps
kbuildsycoca4(3826) KBuildSycoca::checkTimestamps: timestamps check ok
kbuildsycoca4(3826) kdemain: Emitting notifyDatabaseChanged ()
ksnapshot: Fatal IO error: client killed

[1]    Exit 1                    ksnapshot
[bob@pcbsd-2823] ~% ksnapshot &
[1] 3915
[bob@pcbsd-2823] ~% 
```

FIGURE 22.4 The X Window System FVWM screen display.

pop-up or cascading pull-down menus (function I in Table 22.1). Typically, these menus fall into three general categories. Depending on how FVWM has been configured at installation, one button may present a cascading pull-down menu of predefined client applications that you can run by making a menu choice.

An example of the cascading choices found on a typical pull-down menu of this type is shown in Figure 22.5. The cascading menus found in this figure are activated by pressing and holding down the left-most mouse button when the cursor is in the root window.

Hint: At the bottom of the root menu, there is a choice to **Exit Fvwm**. When you make this choice, another fly-out menu of choices is presented that allows you to **Restart Fvwm**, so you don't have to completely log off the system to restart FVWM!

The middle mouse button displays a menu of window modification operations that you can perform by making a menu choice. Typically these modification operations are move, resize/reproportion, raise in the stack to expose the window, lower the window in the stack, (de)iconify the window, (un)maximize the window, and destroy or close the window.

The right-most mouse button displays a list of all open windows, and allows you to bring any of them to the top of the stack of windows displayed on the screen and make that window the current window of operation.

The (de)iconify window function is accomplished by use of the button labeled A in Figure 22.4, and it is found in the title bar of the frame surrounding a client application window. Clicking the left-most mouse button on this screen button reduces the window to

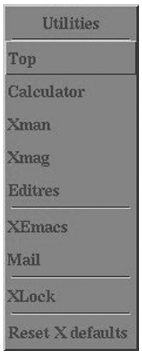
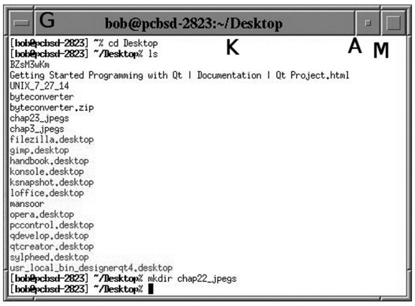


FIGURE 22.5 Pull-down menu to launch utilities.

an icon, which is shown and also labeled A in the figure. Each client application window is surrounded by a frame containing several window manager components that allow the user to perform function G (Modify window) from Table 22.1.

Labeled C in Figure 22.4 is a client application window (an *xterminal*, or *xterm* for short), which provides a CUI to the operating system of the computer this server is linked to by default. Some window modification components surrounding an xterm window are further described in Figure 22.6. The *focus* of the server is sometimes known as the *current position* of the graphics cursor in the screen display, represented by function F in Table 22.1. When the focus of the server is in an xterm window, and the shell prompt appears at the upper-left corner of the window, you are able to type in the UNIX commands you have learned in the previous 20 chapters of this book to have the operating system of the client machine that is controlling this xterm take actions.

A typical virtual screen menu, which performs function H from Table 22.1, is displayed at label H in Figure 22.4. To use this menu of virtual screens, the user simply uses the mouse to position the graphics cursor over one of the tiles in the virtual screen menu, clicks the left-most mouse button, and another portion of the root window is displayed, enabling the user to place other client application windows in that tile. The desktop is not limited by the size of the physical screen display, since a virtual screen display consists of the space that is defined by the area of all the tiles in the virtual screen menu.



```
[bob@pcbsd-2823] ~% cd Desktop
[bob@pcbsd-2823] ~/Desktop% ls
BZsM3wKm
Getting Started Programming with Qt | Documentation | Qt Project.html
UNIX_7_27_14
byteconverter
byteconverter.zip
chap23_jpegs
chap3_jpegs
filezilla.desktop
gimp.desktop
handbook.desktop
konsole.desktop
ksnapshot.desktop
loffice.desktop
mansoor
opera.desktop
pccontrol.desktop
qdevelop.desktop
qtcreator.desktop
sylpheed.desktop
usr_local_bin_designerqt4.desktop
[bob@pcbsd-2823] ~/Desktop% mkdir chap22_jpegs
[bob@pcbsd-2823] ~/Desktop% █
```

FIGURE 22.6 Xterm window and modify components.

The FVWM window modification components included in a frame that is placed around an xterm window are shown in Figure 22.6. These components provide functions A, E, and G from Table 22.1.

The button labeled A in Figure 22.6 is used to iconify or deiconify the xterm window. When one screen tile becomes too cluttered with windows, it is possible to iconify some of the windows to unclutter the screen display.

The button labeled G in Figure 22.6 allows you to perform typical window modification operations. *Be aware that, for many client applications, using the "destroy window" option available with this button does not gracefully terminate the client–server connection.* Usually, within the application user interface provided by the client application; that is, inside the window there is a pull-down menu choice or other action that allows for a graceful exit from the application and a clean disconnect of client from server. There are several dangers inherent in not gracefully exiting an application, foremost of which is that you might lose important data that has not been saved. Also, on networks with software license managers, not gracefully leaving an application does not free up its token to other users of the network, which is bad network etiquette.

The button labeled M in Figure 22.6 allows you to modify the size of the window by *maximizing* the window in the current virtual screen tile. This simply means that you can quickly enlarge the window frame and its contents to take up the entire area of the current tile. To return the window to its original size, click the maximize button again.

The important functional elements not labeled in Figure 22.6 are the *resize handles*, which are activated by moving the graphics cursor to the extreme edges of the frame. They allow you to use the mouse and pointer button to interactively resize and reproportion the window.

There is also another important component of this typical window frame that provides supplemental functions of the window manager. This is the title bar, labeled K in Figure 22.6. A function of the title bar is to allow you to reposition the entire window and its contents by using the pointing button on the mouse.

22.2.4 Customizing the X Window System and FVWM

Now that you are familiar with the appearance and operations of the FVWM window manager, and since the X Window System itself, and FVWM in particular, are highly customizable to suit the interactive needs of a wide range of users, it is worthwhile to know how an individual user can effectively achieve that customization. We will examine three approaches to changing the appearance and functionality of the window system and the window manager. The first approach involves changing the characteristics of applications that run under the X Window System by specifying command line options. The second approach involves modifying or creating an initialization file for the window system and then invoking that file, either by restarting the window system or logging off and then logging back in. The third approach involves modifying or creating an initialization file for the window manager, in our case FVWM, and then invoking that file.

A word of caution is necessary at this point: if you do not know what a modification does to the operation of the window manager or system, don't make it! Certainly before making any modifications to the X Window System environment, become familiar with the default operations that have been set up at installation. Then, make backup copies of any initialization files you want to change, modify the files, and if unexpected behavior results, you can always return to the defaults.

22.2.4.1 Command Line Changes to the X Window System Application

Once you have seen and worked with the default operations of a particular application, it is possible to run that application with customized display characteristics by typing a command along with the appropriate options and arguments. In this section we will modify the display and operating characteristics of the xterm terminal emulator window, using the xterm command and its options. We will also run three other applications using command line options and arguments. A brief description of the xterm command follows:

SYNTAX

```
xterm [[+][-]toolkitoption ...] [[+][-]option ...]
```

Purpose: Run a terminal emulation program in its own window to allow you to type UNIX commands. The + adds the option, the - subtracts the option.

Output: A window with display characteristics determined by **`toolkitoption`** and options.

Commonly used options/features:

`-ah`	This option indicates that xterm should always highlight the text cursor.
`-e program [ arguments ... ]`	This option specifies the program (and its command line arguments) to be run in the xterm window.
`-sb`	This option indicates that some number of lines that are scrolled off the top of the window should be saved and that a scrollbar should be displayed so that those lines can be viewed.

For example, in order to affect the kind of shell that is run in the xterm window when it starts up, you would type the following at the command line:

```
$ xterm -ls &
```

This option indicates that the shell that is started in the xterm window will be a login shell. In order to start an xterm window with an ordinary subshell running in it, you would type the following at the command line:

```
$ xterm +ls &
```

To have the window manager start the xterm window with a scroll bar, which would allow you to scroll backward or forward through the text displayed on screen and retained in a buffer, type the following:

```
$ xterm -sb &
```

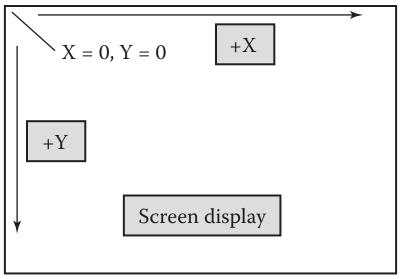


FIGURE 22.7 Screen coordinate system.

A more complete listing of toolkit options and options for the xterm command is given in the man page on your system for xterm.

The following session shows how to run other applications using command line options. The three applications we run—xclock, xbiff, and xterm—are sized and positioned with the `-geometry` command line option and its arguments, which are the same for all three applications. Xclock displays an analog time-of-day clock on the screen as an icon. Xterm opens a CUI to the UNIX operating system. In order to size and position the application windows on your display screen, you must be aware of the way in which the screen coordinates are derived. The coordinate system for screen locations in the X Window System is shown in Figure 22.7.

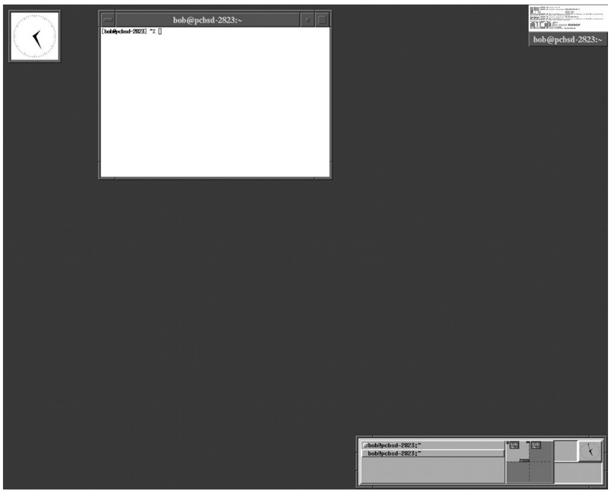
Notice that the origin, 0 in X and 0 in Y, is in the upper-left corner of the screen, and the direction of increasing X is to the right and increasing Y is down. Of course the screen resolution of your display, in other words how many pixels in X and Y can be addressed, is dependent on what kind of monitor and graphics card you have available, and what your X Window System preferences are set to. To run the applications, in an xterm window type the following:

```
$ xclock -geometry 100x100+10+10 &
$ xterm -geometry 80x24+200+10 &
$
```

The upper-left corner of Figure 22.8 shows the appearance and relative size and position of each of the application windows after the previous two commands have been typed in. The arguments for the `-geometry` option are as follows: X-pixel size of window, Y-pixel size of window, X-position of upper-left corner of window, Y-position of upper-left corner of window. So the line `xclock -geometry 100x100+10+10` sizes the xclock to be 100 by 100 pixels, and positions its upper-left corner at the coordinates X = 10, Y = 10 relative to 0,0.

To close down an X Window System application gracefully, you can use the client application mechanisms, which might be a pull-down menu choice or a button press. To close xclock or xbiff, you can find the PID of each by using the `ps` command, and then issue a kill signal for that PID. For example, if xclock had a PID of 904, as shown in the output from `ps`, then typing the following command in an xterm window would close the xclock application:

```
$ kill -9 904
```



```
[bob@pcbsd-2823] ~%
```

FIGURE 22.8 Applications run with command line options.

To find out more about the options for xclock and xterm, see the appropriate UNIX man pages on your system.

EXERCISE 22.5

How do you run an xclock sized at 75×75 located at 200,200?

EXERCISE 22.6

If you run an xterm with no arguments, what is the default window size of the display on screen?

EXERCISE 22.7

After consulting the man page for xterm, how do you run an xterm with background color set to green?

EXERCISE 22.8

How do you run an xterm with the scroll bars disabled or enabled?

22.2.4.2 How to Customize the FVWM Window Manager

Many of the operations, directives, and commands found in the following sections are described in much more detail on the man page for FVWM on your system. We refer you to this particular reference because it gives an easy-to-follow, numbered table of contents to the entire man page for FVWM that organizes the entries into a more useful structure.

22.2.4.2.1 Introduction It is possible to make specific, personalized changes to the look, feel, and most importantly, the functionality and operability of the FVWM window manager as it runs in any user's individual account on a UNIX system.

To accomplish this individualized customization of your account on the system, do the following simple steps. These steps were done on PC-BSD, and reflect the configuration files and the path to it on that system. If you are using another UNIX system, file specifications and paths will vary according to your system, but will be very similar. Also, if you do not change the default configuration file that FVWM uses to configure itself, you can always go back to using that one if your customization efforts do not work. Thus you can use the default configuration file as your backup!

Step 1: If you do not have a **.fvwm** directory in your home directory, create an empty one. If you do have one, putting a file named **config** in it, as shown in the following steps, will allow you to customize FVWM. If you already have files such as **.fvwmrc** or **.fvwm2rc** in the **.fvwm** directory, disable those by renaming them to anything but **config**.

Step 2: Copy the system example configuration file for FVWM, named **system.fvwm2rc**, from **/usr/local/etc/system.fvwm2rc** to the directory **~/.fvwm**, rename the file **config**, and give yourself execute privilege on that file.

Step 3: Use your preferred text editor to make customization changes in the file according to what is shown in the following sections. After making changes, you can start or restart FVWM and your changes will take effect.

EXERCISE 22.9

Do the three preceding steps so you have an executable version of the **~/.fvwm/config** file. Then start or restart FVWM as shown in the hint in Section 22.2.3.2, and as you go through the following sections, you will be presented with examples of the coded directives and commands in that config file.

The structure of coded directives and commands that are found in the config file is referred to and explained in the sections that follow, and is summarized in Table 22.2. In general, these directives and commands dictate the full extent to which customization can

TABLE 22.2 FVWM Config File Directives and Commands

Directives or Commands	**What It Does**
Global Settings	Changes colors and fonts used in window borders and menus
Functions	Focus and icon placement
Bindings	Sets the style of the virtual desktop and pager
Window Decor	Sets the paths to modules and icons
Menu	Sets the window styles and decorations, such as width of borders
Modules	Define functions bound to mouse and mouse buttons
FvwmTaskbar	Defines user menus shown in the root window

be done. We provide additional example code for any particular grouping of directives or commands that is not found in the sample config file.

Commented lines in the config file begin with the pound sign (#); therefore, placing a # before a line in the file turns the directive or command found on that line into a comment, thus negating its effect.

22.2.4.2.2 Global Settings These establish environmental variables and desktop setups.

Example lines from **~/.fvwm/config**:

```
EdgeResistance 250 10
EdgeScroll 100 100
ClickTime 750
DeskTopSize 2x2
```

For example, if you do not want all icons to follow you around the virtual desktop into every virtual screen, simply add a # in front of the line which reads `StickyIcons`. This is where the "search and replace" or "find" feature of your text editor comes in handy.

If you want to be able to move into any virtual screen simply by "rolling" the mouse, find the line in your **.fvwmrc** file that reads

```
EdgeScroll 100 100
```

and make sure that line is <u>not</u> commented out. The arguments of the `EdgeScroll` command let you flip through 100 percent of each virtual screen display as you roll the mouse and change the current position into any of the virtual screen tiles. Otherwise, you would have to use the tiled display found in the lower-right hand corner of Figure 22.4, labeled H, and click in the appropriate virtual tile in order to map it into the physical screen coordinates.

22.2.4.2.3 Functions Functions are part of the programming or scripting language in FVWM and allow execution of multiple commands and directives. The following examples are initialization functions.

Example lines from **~/.fvwm/config**:

```
AddToFunc StartFunction
+ I Module FvwmButtons
AddToFunc InitFunction
+ I exec xsetroot -mod 2 2 -fg rgb:55/40/55 -bg rgb:70/50/70
# For some SM-s (like gnome-session) there is an internal
background setter.
AddToFunc SessionInitFunction
+ I Nop
```

To build complex functions into the config file, you must remember to *forward reference* them in the file, which means that you should place them in the file before they are called in any way. The following is an example of a complex function that moves (changes screen placement) or raises (brings to the top of the window stack) a window using mouse movement and pointer/button clicks.

```
Function "Move-or-Raise"
      Move  "Motion"
      Raise "Motion"
      Raise "Click"
      RaiseLower  "DoubleClick"
EndFunction
```

22.2.4.2.4 Bindings Bindings associate either a key or mouse button with an action.

Example lines from **~/.fvwm/config**:

```
# some simple default key bindings:
Key Next          A       SCM     Next (AcceptsFocus) Focus
Key Prior         A       SCM     Prev (AcceptsFocus) Focus
# some simple default mouse bindings:
#   for the root window:
Mouse 1     R     A     Menu RootMenu Nop
Mouse 2     R     A     Menu Window-Ops Nop
Mouse 3     R     A     WindowList
```

The most useful aspect of window manager customization is being able to define your own menus to be activated by the mouse buttons when the current position is located in a specific place on screen. For example, when the current position is in the root window, the pop-up menus **Utilities**, **Window Ops**, and **winlist** are activated by, or *bound* to, the mouse buttons. This is accomplished by the following three lines of code in the config file:

```
Mouse 1     R     A     Popup "Utilities"
Mouse 2     R     A     Popup "Window Ops"
Mouse 3     R     A     Module "winlist" Fvwm2Winlist transient
```

These menus are made up of a collection of menu choices that either run applications or generate other cascading submenu choices. For example, the module **winlist** calls on a resource to display a list of all open windows on screen. The R on each line means that this menu is activated in the root window. The A on each line means that you can use any keystroke modifier in addition to pressing the mouse button.

22.2.4.2.5 Window Decor This defines the look of the windows.

Example lines from **~/.fvwm/config**:

```
AddToDecor ExampleDecor
+ TitleStyle LeftJustified Height 18
+ ButtonStyle 1 ActiveUp Vector 4 30x30@3 60x60@3 60x30@4 30x60@3
  -- Flat
+ ButtonStyle 1 ActiveDown Vector 4 30x30@3 60x60@3 60x30@4
  30x60@3 -- Flat
+ ButtonStyle 1 Inactive Vector 4 30x30@3 60x60@3 60x30@4 30x60@3
  -- Flat
+ ButtonStyle 3 ActiveUp Vector 5 30x60@3 60x60@3 60x50@3 30x50@3
  30x60@3 -- Flat
+ ButtonStyle 3 ActiveDown Vector 5 30x60@3 60x60@3 60x50@3
  30x50@3 30x60@3 -- Flat
+ ButtonStyle 3 Inactive Vector 5 30x60@3 60x60@3 60x50@3 30x50@3
  30x60@3 -- Flat
+ ButtonStyle 5 ActiveUp Vector 7 30x30@3 30x60@3 60x60@3 60x30@3
  30x30@3 30x35@3 60x35@3 -- Flat
+ ButtonStyle 5 ActiveDown Vector 7 30x30@3 30x60@3 60x60@3
  60x30@3 30x30@3 30x35@3 60x35@3 -- Flat
+ ButtonStyle 5 Inactive Vector 7 30x30@3 30x60@3 60x60@3 60x30@3
  30x30@3 30x35@3 60x35@3 -- Flat
+ TitleStyle -- Flat
+ BorderStyle Simple -- NoInset Flat
+ ButtonStyle All -- UseTitleStyle
```

22.2.4.2.6 Menus These are lists or pop-ups on a predefined taskbar, or one of your own design!

Example lines from **~/.fvwm/config**:

```
AddToMenu RootMenu    "Root Menu"     Title
+                     "XTerm"               Exec exec xterm
+                     "Rxvt"                Exec exec rxvt
+                     ""              Nop
+                     "Remote Logins"       Popup Remote-Logins
+                     ""              Nop
+                                     "Utilities" Popup Utilities
+                     ""              Nop
+                     "Fvwm Modules"        Popup Module-Popup
+                     "Fvwm Window Ops"     Popup Window-Ops
+                     "Fvwm Simple Config Ops" Popup Misc-Ops
+                     ""              Nop
+                                      "Refresh Screen" Refresh
+                                      "Recapture Screen" Recapture
+                     ""              Nop
+                     "Exit Fvwm"     Popup Quit-Verify
```

A useful menu addition:

In order to define your own menu, the following example shows a customized menu definition which can be bound to the right-most mouse button. You can add this menu definition at the end of the menu section in the sample config file:

```
# ProgramMenu
# A collection of three useful programs activated with the right-
most mouse button
AddToMenu ProgramMenu "Useful Programs" Title
+      "GNU emacs" Exec exec emacs
+      "Dolphin"   Exec exec dolphin
+      "FileZilla" Exec exec filezilla
```

A description of each element of the menu definition is as follows:

After the comments explaining the contents of the menu in brief, any menu definition must begin with a line that has the command `AddToMenu`. A handle that can be referenced by other menus or components like mouse buttons follows the command; in other words, if you wish to call this menu as a submenu somewhere else in the config file, you would refer to it with the name `ProgramMenu`. Next, the title **Useful Programs**, which will appear as text at the top of the menu, is placed on its own line. A plus (+) symbol appears as the first character in each line on which you want a menu choice to appear. Then the title for the menu choice appears on each line, such as **Dolphin**. The three programs we are adding to this menu, emacs, dolphin, and filezilla, are placed after three `Exec exec` commands, which follow the entry that will appear in the menu for each. Table 22.3 shows how this menu will appear on screen.

To actually bind this menu to the right-most mouse button, and have it activated when the current position is in the root window, the following lines must appear in the config file where you bind mouse buttons. Notice that the previous binding of the **winlist** module to this mouse button has been commented out of the file!

```
#Mouse 3     R      A      winlist
Mouse 3      R      A      ProgramMenu
```

EXERCISE 22.10

Using the preceding menu example as a model, design your own menu activated by the right-most mouse button to launch some of the important applications you run on your system. Name the menu **MyApplications**. Hint: A program that does not open its own window when it begins to execute will not show anything or launch!

TABLE 22.3 Your Own Pop-Up Menu

Useful Programs
GNU emacs
Dolphin
FileZilla

EXERCISE 22.11

In the sample config file, change the default program launched by the **Utilities** menu from Xemacs to emacs. Test your modification by restarting FVWM and making the new **Utilities** choice emacs.

22.2.4.2.7 Modules A module in FVWM is a user-written program that runs as a separate UNIX process but passes commands to FVWM to execute. Modules work by transmitting commands to FVWM through UNIX IPC.

Example lines from **~/.fvwm/config**:

```
# Establish the Colorset
Colorset 27 fg rgb:00/00/00, hi rgb:00/00/00, sh rgb:00/00/00, bg
rgb:e9/e9/d9
#Define the Module
DestroyModuleConfig FvwmIdent: *
*FvwmIdent: Colorset 27
*FvwmIdent: Font "xft:Sans:Bold:size=12:antialias=True"
```

22.2.4.2.8 FvwmTaskBar This is the bar at the bottom-right of the FVWM root window, as seen in Figure 22.4

Example lines from **~/.fvwm/config**:

```
Style "FvwmTaskBar" NoTitle, !Handles, !Borders, Sticky,
WindowListSkip, \
  CirculateSkip, StaysOnBottom, FixedPosition, FixedSize,
!Iconifiable
DestroyModuleConfig FvwmTaskBar: *
*FvwmTaskBar: Geometry +0-0
*FvwmTaskBar: Rows 3
*FvwmTaskBar: Font "xft:Sans:Bold:pixelsize=12:minispace=True:ant
                    ialias=True"
*FvwmTaskBar: SelFont "xft:Sans:Bold:pixelsize=12:minispace=True:
                       antialias=True"
*FvwmTaskBar: StatusFont "xft:Sans:Bold:pixelsize=12:minispace=Tr
                          ue:antialias=True"
*FvwmTaskBar: Colorset 9
*FvwmTaskBar: IconColorset 9
*FvwmTaskBar: FocusColorset 9
*FvwmTaskBar: TipsColorset 9
*FvwmTaskBar: UseSkipList
*FvwmTaskBar: UseIconNames
*FvwmTaskBar: ShowTips
*FvwmTaskBar: StartName FVWM
*FvwmTaskBar: StartMenu FvwmRootMenu
```

```
*FvwmTaskBar: Button Title ATerm, Icon mini/xterm.png, Action
              (Mouse 1) FvwmATerm
*FvwmTaskBar: Action Click3 Menu FvwmWindowOpsMenu
*FvwmTaskBar: StartIcon mini/fvwm.png
```

22.2.4.2.9 FvwmPager

FvwmPager is used to show the layout of your virtual desktop. It will show all the pages and desktops you have set up, and the windows opened in each.

Example lines from **~/.fvwm/config**:

```
*FvwmPager: Back #908090
*FvwmPager: Fore #484048
#*FvwmPager: Font -adobe-helvetica-bold-r-*-*-10-*-*-*-*-*-*-*
# turn off desktop names for swallowing in the previous button bar
example:
*FvwmPager: Font none
*FvwmPager: Hilight #cab3ca
*FvwmPager: Geometry -1-1
*FvwmPager: Label 0 Misc
*FvwmPager: Label 1 Maker
*FvwmPager: Label 2 Mail
*FvwmPager: Label 3 Matlab
*FvwmPager: SmallFont 5x8
*FvwmPager: Balloons                All
*FvwmPager: BalloonBack             Yellow
*FvwmPager: BalloonFore             Black
*FvwmPager: BalloonFont             lucidasanstypewriter-12
*FvwmPager: BalloonYOffset          +2
*FvwmPager: BalloonBorderWidth      1
*FvwmPager: BalloonBorderColor      Black
```

22.2.4.2.10 FvwmButtons Buttons and button bars can create freeform panels of any size or shape (including nonrectangular windows). It can swallow other applications, even applications not designed for docking, have "panels" that slide out and consist of other panels, has a "startup notification" feature, hundreds of possible bindings, uses the FVWM Colorsets feature, and can alter dynamically and respond to window manager events.

Example lines from **~/.fvwm/config**:

```
################## FvwmButtons button-bar ########################
*FvwmButtons: Geometry 520x100-1-1
*FvwmButtons: Back bisque3
*FvwmButtons: (Frame 2 Padding 2 2 Container(Rows 2 Columns 5
Frame 1 \
Padding 10 0))
```

```
*FvwmButtons: (3x2 Frame 2 Swallow "FvwmIconMan" "Module
FvwmIconMan")
*FvwmButtons: (1x2 Frame 2 Swallow(UseOld) "FvwmPager" "Module
FvwmPager 0 0")
*FvwmButtons: (1x2 Frame 0 Container(Rows 2 Columns 2 Frame 0))
*FvwmButtons: (Frame 2 Swallow(UseOld,NoHints,Respawn) "xbiff"
'Exec exec xbiff -bg bisque3')
*FvwmButtons: (Frame 3 Swallow(UseOld,NoHints,Respawn) "xclock"
'Exec exec xclock -bg bisque3 -fg black -hd black -hl black
-padding 0 -update 1')
*FvwmButtons: (2x1 Frame 2 Swallow(UseOld,NoHints,Respawn) "xload"
'Exec exec xload -bg bisque3 -fg black -update 5 -nolabel')
*FvwmButtons: (End)
*FvwmButtons: (End)
```

For a more detailed description of the options available for customizing FVWM, see the man page for FVWM.

To gain more familiarity with the features and utilities of a nonintegrated GUI installation, go to Problems 1–12 at the end of this chapter.

22.3 THE KDE4 DESKTOP MANAGER

The default desktop management system installed with PC-BSD is KDE4.

KDE stands for the K Desktop Environment, and was developed by a volunteer organization of many programmers. The KDE4 desktop manager is an *integrated system*, in the sense that it provides a consistent and uniform implementation of functions, such as an application programmers interface (API), an object request broker (ORB), window management, desktop configuration tools, session management, and most importantly, application programs. The uniformity of these functions in an integrated system necessarily goes beyond the rudimentary provisions that the X Window System makes for creating and maintaining a graphical interface to UNIX. The drawbacks to this kind of system are its size and complexity, not making it feasible, for example, for installation on an embedded system with limited memory and disk capacity. In the sections that follow, we assume that you have installed PC-BSD as your UNIX system. The default installation method on PC-BSD specifies that the KDE4 desktop manager will start automatically.

If your particular UNIX system does not start KDE4 automatically when the system boots up, but you have installed KDE4 as your default GUI, you can always begin a KDE4 session by typing `startx` at the UNIX shell prompt after you have logged in. We do not cover the installation of KDE4 as a package on any UNIX system, but leave this as a problem at the end of the chapter.

22.3.1 Logging In and Out

After successfully installing and after the first login process to your UNIX system, every subsequent login is done by using a login window, similar to one of the methods shown in

Chapter 2, Section 2.3. Depending on which integrated system and window manager you have designated as the default when you or your system administrator installed UNIX, you will see a login dialog box similar to the one provided in PC-BSD. We show a KDE login window in Figure 22.9, which is similar to what you saw in Chapter 2, Section 2.3. You can now enter your username and password into the login dialog box. This dialog box also allows you to make other important system choices, such as changing the type of window manager you will use, rebooting the operating system, or halting the operating system in preparation for powering down the hardware of the computer. Most ordinary users of UNIX on a network will only log in and out using this dialog box. If you have UNIX running on a stand-alone computer, you will sometimes have to restart, or *reboot*, the computer using the other dialog box choices. When halting the system, it is always a good idea to allow UNIX to completely "unload" itself before turning off the power to the computer.

After you have successfully logged into a KDE4 session, your screen display should look similar to Figure 22.10.

The KDE4 desktop has a very similar look and feel to many other desktop systems that you might be familiar with, such as those found in OS X or a Windows computer system.

The difference between KDE4 and OS X or Windows operation is that when pointing and clicking to launch a program in KDE4, you use a single click of the left-most mouse button to accomplish the launch.

Looking at Figure 22.10, the first similarity you might notice is the grouping of pictures in the bar at the bottom of the screen display. This bar is known as the *Panel*, and it performs as an information center and "launchpad" for many of the desktop's facilities and application programs. Most importantly, there are two open windows on the screen display, one with the title bar heading `bob-Dolphin`, and the other with the title bar heading `bob:  csh-Konsole`. The first window is a KDE4 application window that serves as the default file manager, and is known as Dolphin. The second window is an xterminal, or xterm for short, that allows you to type UNIX commands and have the computer take actions based upon the commands, similar to the console you would work in if you did not have a GUI to UNIX.

At this point, these two open windows allow you to do file management in UNIX with a GUI file manager (Dolphin) or, more traditionally, with a CUI using the text-based commands for file management in a console or xterm window.

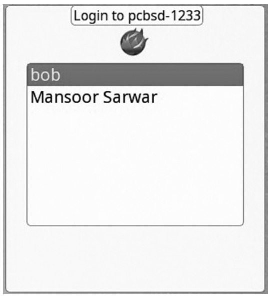
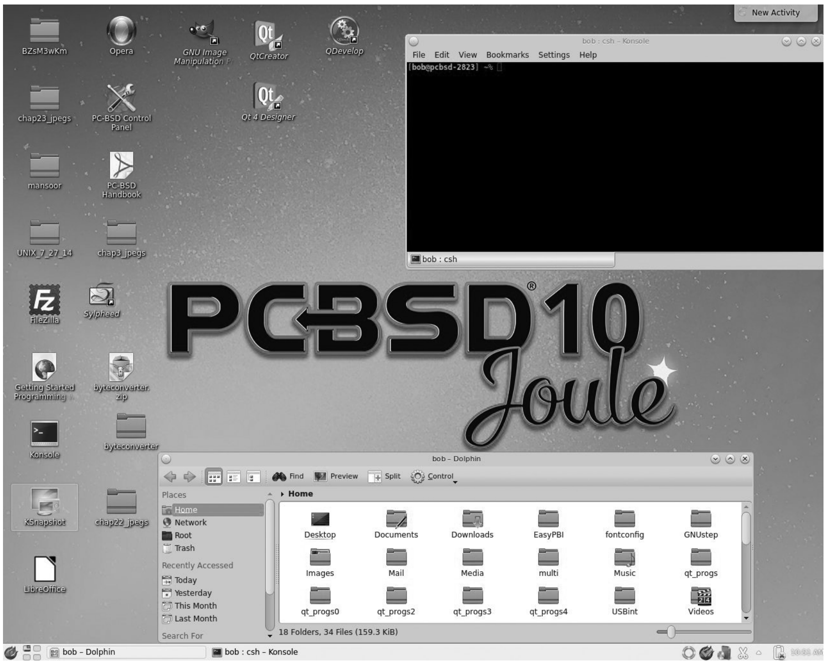


FIGURE 22.9 KDE4 login window.



```
[bob@pcbsd-2823] ~%
```

FIGURE 22.10 The KDE4 screen display.

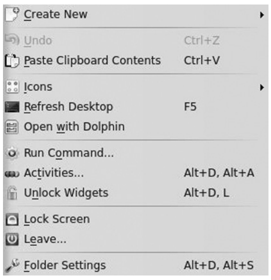


FIGURE 22.11 Right-click menu.

If you click and hold down the right-most mouse button (assuming you have a three-button mouse) when the cursor is in the background area of the desktop, a menu appears allowing you to accomplish some common tasks, such as create a new folder on the desktop, or view and edit desktop icon properties. Figure 22.11 shows this menu.

At this point, if you wanted to log out of the current session, or take other system actions such as reboot or halt the UNIX operating system, you would make the **Leave** menu choice, as seen near the bottom of Figure 22.11. You could also use the Kickoff Application Launcher icon found in the extreme lower-left corner of the screen display in Figure 22.10. If you left-click on this icon, a series of choices appears as a pop-up menu, and the **Leave** choice is on this menu. In the following sections, we will describe the components of the KDE4 desktop.

22.3.2 The KDE4 Panel

By far the most important component on the KDE4 desktop is the Panel.

Referring to Figure 22.10, the default display of the Panel is across the bottom of the screen display. The Panel components, shown in Figure 22.12 from left to right, are named and briefly described in Table 22.4.

The most important component of the Panel is the Kickoff Application Launcher, the left-most component in Figure 22.12. If you left-click on it, the Main Menu, as seen in Figure 22.13, appears, and contains fly-out menus enabling you to launch a preset list of applications and utilities. For example, as seen in Figure 22.14, the **Applications>Utilities** fly-out menu of the Main Menu contains a large listing of useful utilities which can be launched from it. There are a number of other fly-out menus activated from the **Applications>Utilities** menu that allow you to launch user applications, graphics applications, Internet tools, multimedia tools, and other system-wide applications and development applications. Finally, there are two buttons on the Main Menu, one of which allows you to lock the screen with password protection so it cannot be tampered with when you are away from the display, and the other to logout.

Another extremely important and signature component of the KDE4 Panel is the Mount Tray: item F in Table 22.4 and seen on the right side of Figure 22.12. This icon allows you to graphically manage removable media, such as USB thumb drives or external USB hard drives, on your system. When a usable USB thumb drive is inserted into one of the computer's USB ports, the Mount Tray indicates that it is attached (meaning it shows up in a listing of **/dev**), and either is automatically mounted or can be manually mounted via the Mount Tray. This signature graphical management feature, above all others, clearly differentiates twenty-first-century UNIX from twentieth-century UNIX. In 1980s UNIX, you would have to know the exact command line syntax for the `mount` and `umount` commands, and all of the options, option arguments, and command arguments possible for those commands. In 2015, you point and click on a graphical icon.

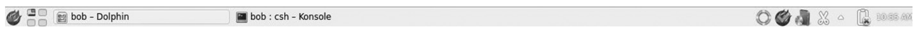


FIGURE 22.12 KDE4 Panel components.

TABLE 22.4 KDE4 Panel Component Descriptions

LtoR	Icon Name	Description
A	Kickoff Application Launcher	Allows you to launch applications and desktop utilities
B	Virtual Window Pager	Allows you to move between virtual tiles of the desktop
C	Open Windows	Displays icons for all open windows, such as Dolphin and Konsole
D	Life Preserver	Allows you to create backups of system and user files
E	System Update Manager	Shows the status of and allows system updates
F	Mount Tray	Allows mounting and unmounting of removable media, such as USB thumb drives
G	System Paste Buffer	Lists contents of system paste buffer
H	Battery Status	Shows status of a battery, if attached
I	Clock/Date	Clock/date display and control

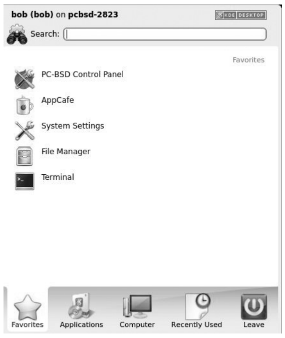


FIGURE 22.13 Main K menu.

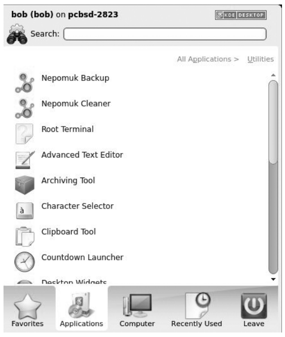


FIGURE 22.14 Utilities submenu of the KDE4 Main Menu.

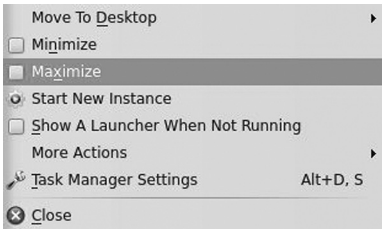


FIGURE 22.15 Right-click menu on a panel object.

Right-clicking on any of the objects in the Panel activates a menu that allows you to manipulate that particular object. For example, if you right-click over any icon in the Panel, you get menu choices that allow you to remove that button from the Panel, move that button to a new location on the Panel, or obtain and change the properties of that application (Figure 22.15).

22.3.3 Adding a Desktop Icon that Launches an Application

One of the most useful aspects of the KDE4 desktop is the ease with which you can reconfigure almost every component of it. The following practice session will show you how to add a new icon to your KDE4 desktop that launches an application. Some of the most useful applications that can be launched by icons are shown in Figure 22.10.

If vim or emacs are not installed as packages on your system, use the AppCafe in PC-BSD (or its equivalent on your UNIX system) to install them.

Practice Session 22.1

Step 1: From the Kickoff Application Launcher Main Menu, make the fly-out menu choices **Applications>Utilities**.

Step 2: Scroll down to the icon that represents the vim editor.

Step 3: Hold down the left-most mouse button when the cursor is over the vim editor icon, and drag the cursor to any place on the main window of KDE4. Release the left-most mouse button. A screen menu appears with the choices **Copy Here**, **Link Here**, and **Cancel**. Click the left-most mouse button on the **Link Here** choice.

Step 4: The same icon for vim that was in the `Applications>Utilities` window is now an icon on the desktop.

Step 5: Click the left-most mouse button on the vim icon on the desktop. The vim text editor launches.

EXERCISE 22.12

Add any application icon choices that you like from the Applications menu to the desktop, using the methods described in Practice Session 22.1.

EXERCISE 22.13

Add emacs as a desktop icon using the method in Practice Session 22.1.

22.3.4 KDE4 Window Manager

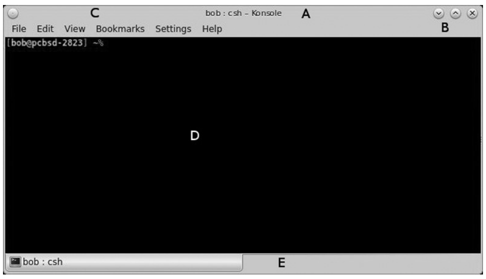
The program which you most directly work with in the KDE4 desktop environment is the window manager. The appearance and interactivity of KDE4 windows, from simple terminal windows to application windows, is controlled by the window manager. The default window management system is called Plasma. There are several other desktop/window managers available to you, the most popular being Awesome, Cinnamon, Fluxbox, Mate, and FVWM. For more information on these window managers, see the Web Resources section at the end of this chapter.

The default KDE4 window manager gives a window "dressing," or standard interactive techniques and their related features, to expedite your work within that window. For example, a scroll bar is provided in a Konsole window to allow you to graphically scroll backward or forward through the text that has been displayed on the screen. For more examples of these features, see Table 22.5 and the graphical references to Figure 22.16.

The window manager provides a frame within which the application can display its graphical output and the user can interact with the application program through user-generated events. The look (appearance of buttons, style, and color; i.e., its *theme*) and feel

TABLE 22.5 Konsole Window Components of a KDE4 Window

Item	Name	Description
A	Title bar	Shows current path and allows window movement
B	Control buttons	Minimize, maximize, and kill window
C	Toolbar	Allows you to use xterminal-specific tools
D	Body	Xterminal CUI console display, where you type commands and see the results
E	Border	Allows resizing of window with "handles"



```
[bob@pcbsd-2823] ~%
```

FIGURE 22.16 Konsole display of an xterminal window.

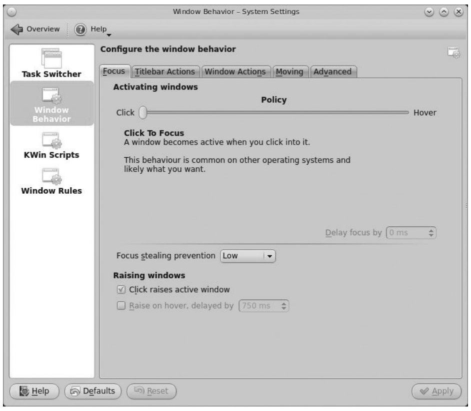
(how buttons work, how menus are activated) of the window manager is independent of the look and feel of the application running in area E as seen in Figure 22.16. Because of this, it is possible to reconfigure the appearance, and also the interactivity, of the default KDE4 window manager, and not affect the look or interactivity of any application running inside of the window manager's border.

To understand the steps necessary to reconfigure the default KDE4 window manager so that it has a different look and feel, do Practice Session 22.3. In this practice session, you will change a feature of the style of interaction with KDE4 known as the *focus policy*. The focus policy is the response of the graphical server to locations of the current position. For example, if you move the mouse, and the on-screen cursor rolls into a window, that window becomes the current window for input and output events. You will also change the appearance of KDE4 to another theme. We assume that you are using the default KDE4 window manager, and it has the "Default Theme" look as the default. We also assume that your virtual desktop tile is one in the virtual screen display.

Practice Session 22.3

Step 0: As preliminary preparation for this session, observe the response of KDE4 as you move the mouse and the screen cursor shifts between open windows, such as a Konsole window or the Dolphin file manager window. As you roll the cursor with the mouse over a window, what behavior does the window display?

Step 1: From the Kickoff Application Launcher, make the menu choice **System Settings**. The `System Settings` window opens on screen. Click on the **`Window Behavior`** icon, and make the **`Window Behavior`** sidebar choice. The KDE4 Window Behavior – System Settings window opens on screen, similar to Figure 22.17.



Configure the window behavior

Activating windows

Policy

Click — Hover

Click To Focus
A window becomes active when you click into it.

This behaviour is common on other operating systems and likely what you want.

Delay focus by 0 ms

Focus stealing prevention Low

Raising windows

☑ Click raises active window

☐ Raise on hover, delayed by 750 ms

FIGURE 22.17 `Window Behavior – System Settings` window.

Step 2: Click on the `Focus` tab. Move the slider all the way to the right, to the `Hover` pole.

Step 3: Left-click on the **`Apply`** button in the `Window Behavior – System Settings` window.

Step 4: Now move the mouse so that the cursor shifts between windows on screen. The behavior of the window manager has changed, depending on what you set the focus policy to be.

Step 5: Left-click on the **`Overview`** arrow in the upper-left corner of the `Window Behavior – System Settings` window. This returns you to the main window, as seen in Step 1.

Step 6: From the Main Menu, click on the **`Workspace Appearance`** icon. Click on the **`Desktop Themes`** sidebar choice. The `Desktop Theme – System Settings` window appears on screen, similar to Figure 22.18.

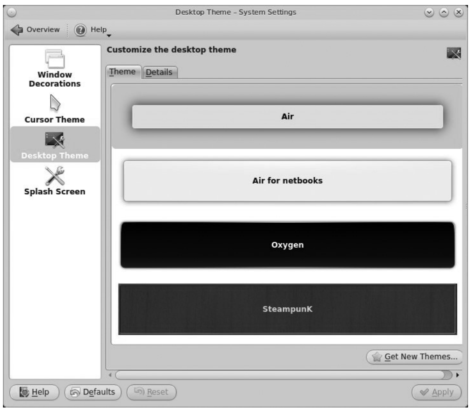


FIGURE 22.18 KDE4 theme manager window showing the default theme.

In this figure, we have already added the "SteampunK" theme to the available choices ("Air," "Air for netbooks," "Oxygen").

Step 7: Left-click once on the **`Get New Themes...`** icon in the lower left of this window. Search for a new theme and install it. It now appears in the `Desktop Theme – System Settings` window. Click on this newly installed theme to make it the current theme.

Step 8: Left-click on the **`Apply`** button at the bottom of the `Desktop Theme – System Settings` window, and if you do not want to test the appearance of other themes on your desktop, then click on the **`Quit`** button at the top of the `System Settings` window. Your desktop now has the theme of your choice!

22.3.5 KDE4 System Settings

The most important configuration tool that you have available to you from the KDE4 desktop is the System Settings facility, activated by making the Kickoff Application Launcher icon **System Settings** choice. This utility allows configuration of single-user attributes, as well as system-wide attributes for all users. A single user with ordinary privileges is allowed to make a variety of changes in the performance of the PC-BSD system itself and the performance of the KDE4 desktop. For example, as shown in Practice Exercise 22.3, you can use a series of on-screen menu choices to change the focus behavior within KDE4 windows and select a new theme.

There are five areas of configuration change groupings in the KDE4 `System Settings` window. By clicking on one of the icons, you can see a configuration display in the right-hand panel associated with that particular group member. For example, if you left-click on the `Application Appearance` icon in **Common Appearance and Behavior**, you have several other icon options available to you: one of these, `Colors`, allows you to colors used in applications. Once you click the `Apply` button at the bottom of the KDE4 System Settings window, that attribute or option change will immediately take effect. To return to the top level of icons, click on the left-facing arrow named `Overview`.

EXERCISE 22.14

Use the KDE4 system settings to change the **Power Management>Screen Energy Saving** to 5 minutes, or to any setting that is convenient for your usage pattern.

EXERCISE 22.15

What KDE4 facility allows you to change the background desktop wallpaper? What is the name of the default file management system used by the KDE4 desktop manager?

22.3.6 KDE4 File Management with Dolphin

To obtain a graphical view of the files on your UNIX system, particularly the files you have in your home and working subdirectories, the Dolphin file manager allows you to quickly find, view, and edit not only the ordinary files themselves, but also the directory structure that contains them. This capability is given to you by the UNIX file maintenance commands that were detailed earlier in this textbook, but Dolphin uses a graphical approach that saves time for the ordinary user, and it is also something that most users are more familiar with. To launch the Dolphin file manager, from the Kickoff Application Launcher, click on the `File Manager` icon. Your screen display of the Dolphin window will look similar to Figure 22.19.

On our PC-BSD UNIX system, by default Dolphin shows the contents of the current working directory, which is the home directory, as seen in Figure 22.19.

There are five important functional areas of the Dolphin window, as shown in Table 22.6.

There are also two other areas of interest in the Dolphin window. In the lower-right corner there is a slider control that allows you to change the displayed size of icons in the main window: slide it to the right and icons get bigger. And above the main window pane, there is a display of the path to the current working directory.

The most useful of the toolbar icons choices is the `Find` icon, which allows you to designate a place in the file structure of the system at which to start the search, and allows you to designate a file name pattern to search for. This utility is very similar to the text-based UNIX `find` command, but much simpler to use. For example, if you wanted to find all

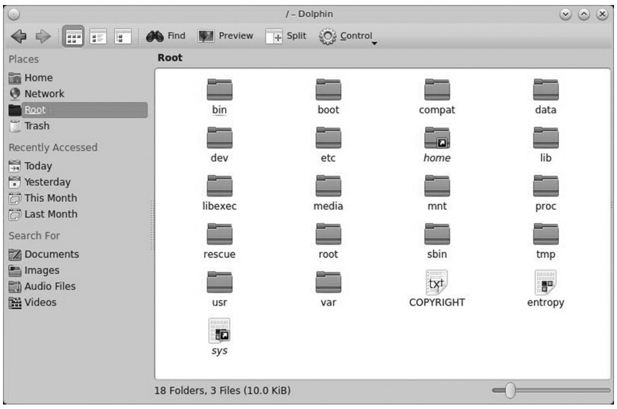
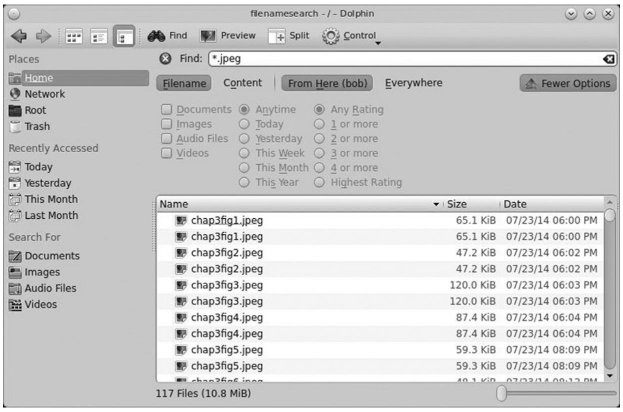


FIGURE 22.19 Dolphin file manager window.

TABLE 22.6 Dolphin Window

Area of Dolphin Window	Location and Description
Title bar	Across the top of the window; window controls like iconify, destroy, etc.
Toolbar icons	At the top; allows you to take actions, most important the **Find** function indicated with the “binoculars” icon, and allows you to change the appearance of file displays very quickly.
Places pane	Along the left side of the screen display; shows you the folders **Home**, **Network**, **Root**, and **Trash**, and recently accessed documents, etc.
Main window pane	Right in the middle; shows you icons by default of files in the current directory.
Status bar	Along the bottom of the window; contains general information about whatever is at the current cursor position.



Find: *.jpeg

Name	Size	Date
chap3fig1.jpeg	65.1 KiB	07/23/14 06:00 PM
chap3fig1.jpeg	65.1 KiB	07/23/14 06:00 PM
chap3fig2.jpeg	47.2 KiB	07/23/14 06:02 PM
chap3fig2.jpeg	47.2 KiB	07/23/14 06:02 PM
chap3fig3.jpeg	120.0 KiB	07/23/14 06:03 PM
chap3fig3.jpeg	120.0 KiB	07/23/14 06:03 PM
chap3fig4.jpeg	87.4 KiB	07/23/14 06:04 PM
chap3fig4.jpeg	87.4 KiB	07/23/14 06:04 PM
chap3fig5.jpeg	59.3 KiB	07/23/14 08:09 PM
chap3fig5.jpeg	59.3 KiB	07/23/14 08:09 PM

117 Files (10.8 MiB)

FIGURE 22.20 Find display in the Dolphin window.

files starting at the home directory that ended in the file extension **.jpeg**, you would click on the `Find` icon and then supply the information in the text bar, similar to what is shown in Figure 22.20. When you click on the `Find` button, you get a view in the main window showing all that match your search criteria or file specification, along the designated path, starting where you specify! A status display shows the progress of your search. If no files are found that match your designation, you are notified as such.

EXERCISE 22.16

Use the Dolphin `Find` icon utility to find all files starting at the root (/) directory that end with the file extension **.bmp**. How many files were found on your system?
(Hint: We found 36 files under the root on our PC-BSD system.) Then, right-click on one of the files to view the path to that image in a new window. To view the image, open it with Gimp or an image viewer of your choice.

EXERCISE 22.17

Use the Dolphin `Find` icon utility to find all files starting in your home directory that begin with the letter M (uppercase).

EXERCISE 22.18

If you have used the Dolphin file manager to find a certain file on the system, how do you change the file's access privileges to execute for user, group, and others on that file from within Dolphin (assuming you own the file or are running Dolphin as root)?

To gain more familiarity with the features and utilities of the KDE4 desktop, go on to Problems 13–19 at the end of this chapter.

22.4 CREATING X WINDOW SYSTEM CLIENT APPLICATION PROGRAMS

The two important points in this section are

- A client application program is made up of two separate parts: a data generation part and a user interface (UI) part, which must work together.
- The basic structure of a client application program is initialization, start an event–request loop, cleanup.

There are two ways to approach creating an X Windows client application program:

- Use a GUI-based *integrated development environment* (IDE; e.g., Qt Designer or Qt Creator) to generate the UI and program data generation code
- Code the graphical interface by programming directly in Qt, GTK, XCB, or the older Xlib, create the program data generation code in C, C++, or Python, and finally combine the GUI and program data generation code

The first way is ostensibly easier. But to create the user interface (UI) component of a client application, and in order to "hook" the data-generating or processing component of your program to the UI, you have to be very familiar with two things:

1. How to do advanced data structure programming in C++, or another available language library interface like Python
2. Knowing the data structure and operability of programming in Qt or GTK+

The second way requires that you know the structure of an X Window System client application program, and if you use Xlib, are familiar with C.

The structure of a client application program is: initialize a connection with the X Server, create an *event loop*, cleanup, and leave. Most of what follows details this structure more fully, and gives some simple examples.

The short way through this section: carefully examine the following examples, and find in them the two important things to remember in those examples. Then you will have a top-down view of how to create client application programs.

We will now show some examples of Qt programming data structure and the program code interface to give you a feel for working in the first way.

22.4.1 Client Application Program Structure and Development Model

In this section, we show where and how a user-written client application program fits into the overall scheme of the components of the X Window System. We then show the simplified structure of such a program. Following from these two illustrations, we detail in a simple and direct fashion how to develop C code for an X Window System client application program.

22.4.1.1 Model Overview

Here are two descriptions of the model.

- First, a picture (Figure 22.21) showing how the components, such as a client application, Xlib, XCB, Qt, or GTK+, the X.org server, and the actual display, are connected via the X Protocol over a network (or not)
- Then a verbal description of the coding process that shapes the model

The X Window System, as seen from a software development model point of view, is a combination of these components.

1. Toolkit IDEs such as Qt and GTK+
2. X client libraries such as XCB and Xlib

   ----
3. The X protocol or display server protocol
4. X display server, X.org server
5. Window manager FVWM

This model is arranged, from top to bottom, in a suitable order for a user writing an application program. The user generates the code either in component 1 or 2; components 3 to 5 are processes that the application program code interprets and executes.

More detailed descriptions of the components are as follows.

1. Toolkit IDEs such as Qt and GTK+ are used to write and put together the code itself, and integrate it with a particular application.

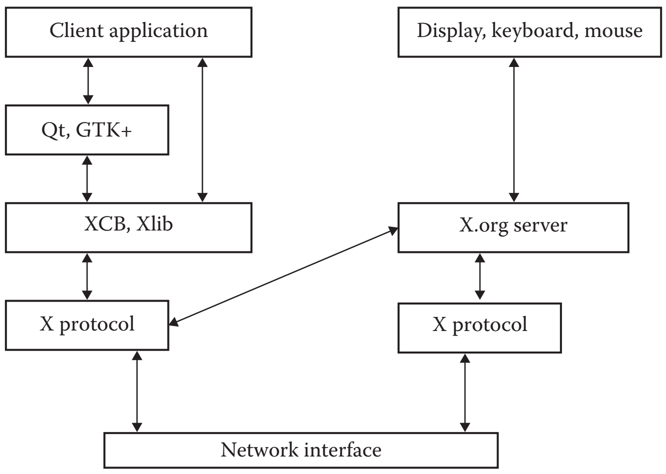


FIGURE 22.21 Client application interaction with other X components.

2. X client libraries are graphical routines or C language bindings that the toolkits generate. The two most important ones are XCB and Xlib. For example, Xlib is a legacy library of more than three hundred utility routines that programmers can use to activate the X protocol. The Xlib utilities are used to accomplish the major tasks in an X Window System user-written application. XCB is the contemporary replacement for Xlib.

3. X clients communicate with X servers (usually, but not necessarily, through a network) using the X protocol. In the X protocol, data is exchanged in an asynchronous manner over a two-way communication channel.

4. The X display server (or X server) is the process executing on a computer and managing the graphics output and input from the computer display (its monitor[s], keyboard, and mouse).

5. The window manager, like any of those used in KDE4 or Gnome, is a program that handles the graphic activities sent to the X display server.

X clients are application programs that use the computer display. The X clients, whether running locally or on a remote computer, send requests to the X server using a communication channel.

The X client application program and the X server program can run on the same machine.

The X protocol software uses a channel between the X client(s) and X server. As long as there is a common networking protocol (e.g., TCP/IP) to provide the channel, the X server can display output from any X client regardless of where it is actually running and the operating system under which the client runs.

With the X protocol running on a particular computer, the X server is listening to the network connections at a specific port and acting on the X protocol requests sent by X clients. The X server manages regions of the screen known as *windows*, where the output from an X client is displayed. When X application programs are running, everything on the screen appears in windows and each window is associated with a specific X client.

Creating a window is one of the basic X protocol requests that an X server handles. The X server considers anything you do with the keyboard and mouse as events to be reported to the X clients. When you press and release a mouse button, the X server sends these input events to the X client that created the window containing the mouse pointer. The X server also sends other kinds of events to X clients. These events inform an X client if anything happens to its window.

X application programs have routines from a toolkit in them. The toolkit may be composed in Qt, which in turn calls Xlib. An X application can also make some direct calls to some Xlib routines for generating text and graphics output in a window.

22.4.1.2 The Structure of a Typical X Client Application Program

A simple description of an X application program would divide it into three major sections:

- Initialization: Open a display that the application can use

- Event loop: Start an event-driven loop that allows the application to communicate with the display
- Cleanup: Clean up and gracefully exit

This description can be further expanded as follows:

- Initialization
    - Perform initialization routines
    - Connect to the X server
    - Perform X-related initialization
- Event loop (while not finished)
    - Receive the next event from the X server
    - Handle the event, possibly sending various drawing requests to the X server
    - If the event was a quit message, exit the loop
- Cleanup
    - Close down the connection to the X server
    - Perform cleanup operations

The initialization section sets up the window system for user interaction. After initialization, the program enters a loop in which it repeatedly tries to get events from the window system and process them.

Finally, before exiting, the program performs any necessary cleanups. Usually the exit code is in a program that is called when the user clicks on the `Exit` button provided by the application (or on the `Close` button on the window frame on the top right of the window).

22.4.1.3 Specifying Resources

All X Window application programs have resource files where options for colors, fonts, and so on can be specified. To take advantage of this capability, programs should be written such that hard coding of resources is avoided, so that resources specified in the configuration files are used.

The resource configuration file for an X application will have the same name as the application specified in the call to `XtAppInitialize()`. It can be located in the directory where the application is launched, or else X will search for this file in the directory specified by the variable `XAPPLRESDIR`.

22.4.1.4 Writing the Code for an X Windows Client Application

Why show three different libraries and methods of writing client application programs? Xlib and XCB are basically procedural programming paradigm libraries, where XCB has a more complex data structure and API than Xlib. Qt uses an *object-oriented client* programming paradigm with attendant data structures and classes, and is coded in C++.

Application code for Xlib or XCB is written in C or C++ and compiled using any of the available C/C++ compilers on the system.

When programming in C++ for Qt, that library has its own facilities to compile, link, and assemble a client application and place it in the context of a Qt project, as shown in Section 22.4.5, "Using the Qt Toolkit."

22.4.2 Xlib versus XCB

The reason we include examples of Xlib and XCB client programs is that they are the two official C libraries for the X Window protocol. Xlib, the predecessor of XCB, was the original X client library, and was the only official X client library until the introduction of XCB. The two libraries are based upon very different schemes: Xlib is a layer further from the X protocol that uses a traditional and programmer-friendly C API, whereas XCB is a very thin layer on top of the X protocol that does not have as transparent and friendly an API. As you can see from our presentation on both Xlib and XCB, the documentation that exists currently for XCB is far less friendly, complete, and descriptive. These two aspects of XCB, documentation and user-friendliness, are a function of its closer relationship with the complex data structure implementation of the X protocol itself.

In practice, the difference in organizing schemes is most evident in how the two libraries handle the fundamental asynchronous event–request model between server and client of the X protocol itself. Xlib attempts to implement the asynchronous X protocol behind a mixed synchronous and asynchronous API, whereas the XCB API is asynchronous.

For example, to lookup the attributes (e.g., size and position) of a window, you would write the following code using Xlib:

```
XWindowAttributes attrs;
XGetWindowAttributes(display, window, &attrs);
/*Execute some code*/
```

The Xlib call to `XGetWindowAttributes()` in the client-side program sends a request to the X server and blocks until it receives a reply from the X server. This is a synchronous request–event sequence.

The following is the code for the same thing in XCB:

```
xcb_get_window_attributes_cookie_t cookie =
        xcb_get_window_attributes(
            connection, window);
/*Execute other code while waiting for the reply from the server*/
xcb_get_window_attributes_reply_t* reply =
        xcb_get_window_attributes_reply(
            connection, cookie, nullptr);
/*Execute some code based on the reply*/
```

```
free(reply);
```

The function `xcb_get_window_attributes` sends the request to the X server, and returns immediately without waiting for the reply. This is an asynchronous request–event sequence. The client program must call `xcb_get_window_attributes_reply` to block on the reply.

The advantage of the asynchronous approach is gained when we need to retrieve the attributes of multiple windows at the same time. Using XCB, we can make multiple requests to the X server at once and then wait for multiple replies. With Xlib, we have to wait for the response to each request before we can send the next one. XCB only blocks for one round-trip network latency period, compared to multiple latency period waits with Xlib.

To be fully asynchronous, the XCB approach leads to a more complex data structure approach, and a less programmer-friendly API. The preceding Xlib code looks like your average C library call; the XCB code has a more complex data structure implementation.

XCB is fully asynchronous, whereas Xlib is not fully synchronous. Xlib has a mixture of synchronous and asynchronous APIs. Functions that do not return values (e.g., `XResizeWindow`, which changes the size of a window) are asynchronous, while functions that return values (e.g., `XGetGeometry`, which returns the size and position of a window) are synchronous. Here is a quote from Volume 1 of the *Xlib Programming Manual* dealing with Xlib's synchronicity:

> Buffering
> Xlib saves up requests instead of sending them to the server immediately, so that the client program can continue running instead of waiting to gain access to the network after every Xlib call. This is possible because most Xlib calls do not require immediate action by the server. This grouping of requests by the client before sending them over the network also increases the performance of most networks, because it makes the network transactions longer and less numerous, reducing the total overhead involved.

22.4.3 Xlib

In this section, we give programming examples using Xlib. It would be useful for you to compare the complexity of the code given here with the code given for XCB programming in the following section. Also, compare the extant documentation for Xlib to the documentation available for XCB.

22.4.3.1 Basic Xlib Theoretical Consideration

Xlib operates on the client–server model, which can be directly contrasted to the traditional networking model of those components. Essentially, the client–server model used for Xlib reverses the role of client and server assumed in the networking model. Below, we give you a practical introduction to programming in the Xlib model.

22.4.3.2 Compiling an Xlib Client Application Program

Be aware that, on a PC-BSD system, you may have to download and install the GNU C compiler (either version 47 or the latest version available) for the following example programs to run, if that has not already been done by your system administrator. If you are the system administrator, you can do this for yourself by using the procedures in the AppCafe in PC-BSD or by using the repository in Solaris.

On our PC-BSD system, we used the following compiler command with the options and arguments shown. On Solaris, you can substitute `gcc` for `gcc47`.

```
gcc47 input_file.c -o output_file -lX11
```

Here, **input_file.c** is the name of your C source code file and **output_file** is the name of the executable program.

22.4.3.3 Sample Xlib Client Application Programs

The following are three elementary sample Xlib programs. Each is preceded by a statement of what the program does and a table listing the X Window functions called along with a description of what the functions do. To get more information about the function calls—for example, what their specific argument list data structure and contents are—consult the online documentation for Xlib given at the end of this chapter.

22.4.3.3.1 Xlib Example Program **test2.c** Objective: Draw a black 500 × 500 pixel window, surrounded by the window manager decorations for whatever window manager you have running on your system. Then, after five seconds, the window will disappear.

Functions called (Table 22.7):

Code:

```
/*Standard includes*/
#include <stdio.h>
#include <stdlib.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/Xos.h>
#include <X11/Xatom.h>
#include <X11/keysym.h>
/*Variable and pointer declarations*/
Display *dis;
```

TABLE 22.7 Xlib Display Functions

Xlib Function Name	Description
`XOpenDisplay`	Returns a display structure that serves as a connection to the X server
`XCreateSimpleWindow`	Creates an unmapped **InputOutput** subwindow for a specified parent window
`XMApWindow`	Maps the window and all its subwindows
`XFlush`	Flushes the output buffer

```
Window win;
int main() {

/*Initialization*/

dis = XOpenDisplay(NULL);
win = XCreateSimpleWindow(dis, RootWindow(dis, 0), 1, 1, 500, 500,
      0, BlackPixel (dis, 0), BlackPixel(dis, 0));
XMapWindow(dis, win);
/*Cleanup and exit*/
XFlush(dis);
/*Sleep 5 seconds before closing.*/
sleep(5);
return(0);
}
```

EXERCISE 22.19

What basic structural component of an X Window System program is missing from the preceding program?

EXERCISE 22.20

What are the arguments supplied to the `XCreateSimpleWindow` function in the preceding program? Consult the Xlib documentation to give a complete listing and description of all arguments.

22.4.3.3.2 Xlib Example Program **test4.c** Objective: Open a window on the display with the title `Report`, and then place the text string `UNIX Rocks` at any mouse click–indicated position in the window. In addition, you can press keyboard keys and they will be echoed on the console screen, telling you what key you pressed. This introduces in a straightforward and easy-to-understand manner the program model for X Window System client applications, and illustrates the concept of a GC that specifies many of the characteristics of windows and other objects.

New functions called (Table 22.8):

Code:

```
/* Xlib and standard C headers */
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/Xos.h>
#include <stdio.h>
#include <stdlib.h>
/* Declare the X variables and pointers*/
Display *dis;
int screen;
```

TABLE 22.8 Xlib Event Functions

Xlib Function Name	Description
`XNextEvent`	Copies the first event from the event queue into the specified `XEvent` structure and then removes it from the queue
`XLookupString`	Translates a key event to a `KeySym` and a string
`XSetForeground`	Sets the foreground attributes of a given GC
`XDrawString`	Draws text characters in a given drawable
`XSetStandardProperties`	Provides a means by which simple applications set the most essential properties with a single call
`XSelectInput`	Requests that the X server report the events associated with the specified event mask
`XCreateGC`	Creates and returns a GC; can be used with any destination drawable with the same root and depth as the specified drawable
`XSetBackground`	Sets the background attributes of a given GC
`XClearWindow`	Clears the entire area in the specified window
`XMapRaised`	Maps the window and all of its subwindows that have had map requests and raises the window to the top of the stack
`XFreeGC`	Destroys the specified GC as well as all the associated storage
`XDestroyWindow`	Destroys the specified window as well as all of its subwindows and causes the X server to generate a `DestroyNotify` event for each window
`XCloseDisplay`	Closes the connection to the X server for the display specified in the display structure and destroys all windows and resource IDs
`XClearWindow`	Clears the entire area in the specified window

```
Window win;
GC gc;
/* X routines */
void init_x();
void close_x();
void redraw();
main () {

/*Initialization*/

    XEvent event; /* declare the XEvent */
    KeySym key;   /* KeyPress Events */
    char text[255];     /* char buffer for KeyPress Events */
    init_x();

/* Start the Event-Request Loop*/

      while(1) {
            /* get the next event.
               We set the mask for events that we want detected
            */
            XNextEvent(dis, &event);
            if (event.type==Expose && event.xexpose.count==0) {
```

```
            /* the window was exposed redraw it! */
                  redraw();
            }
            if (event.type==KeyPress&&
               XLookupString(&event.xkey,text,255,&key,0)==1) {
            /* use the XLookupString routine to convert the invent
               KeyPress data into regular text.
            */
                  if (text[0]=='q') {
                        close_x();
                  }
                  printf("You pressed the %c keyn",text[0]);
            }
            if (event.type==ButtonPress) {
            /* report where the mouse Button was Pressed */
                  int x=event.xbutton.x,
                      y=event.xbutton.y;
                  strcpy(text,"UNIX Rocks");
                  XSetForeground(dis,gc,rand()%event.xbutton.x%255);
                  XDrawString(dis,win,gc,x,y, text, strlen(text));
            }
      }
}
void init_x() {
/* Set the colors black and white */
      unsigned long black,white;

      dis=XOpenDisplay((char *)0);
      screen=DefaultScreen(dis);
      black=BlackPixel(dis,screen),
      white=WhitePixel(dis, screen);
      win=XCreateSimpleWindow(dis,DefaultRootWindow(dis),0,0,
            300, 300, 5,black, white);
      XSetStandardProperties(dis,win,"Report","E",None,NULL,0,NULL);
      XSelectInput(dis, win, ExposureMask|ButtonPressMask|KeyPress
      Mask);
           gc=XCreateGC(dis, win, 0,0);
      XSetBackground(dis,gc,white);
      XSetForeground(dis,gc,black);
      XClearWindow(dis, win);
      XMapRaised(dis, win);
};

void close_x() {

/* Cleanup */

      XFreeGC(dis, gc);
      XDestroyWindow(dis,win);
      XCloseDisplay(dis);
      exit(1);
};

void redraw() {
      XClearWindow(dis, win);
};
```

22.4.3.3.3 Xlib Example Program **test1.c** Objective: Produce a simple window on the display that prints `Hello, World!` and draw a small filled-in rectangle in black. It may be closed by pressing <`Ctrl+C`>.

New functions called (Table 22.9):

Code:

```
/*
  * Simple Xlib application drawing a box in a window.
*/
#include <X11/Xlib.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    Display *display;
    Window window;
    XEvent event;
    char *msg = "Hello, World!";
    int s;

/*Initialization*/

    /* open connection with the server */
    display = XOpenDisplay(NULL);
    if (display == NULL)
    {
        fprintf(stderr, "Cannot open display\n");
        exit(1);
    }
    s = DefaultScreen(display);

    /* create window */
```

TABLE 22.9 Xlib Drawing Function

Xlib Function Name	Description
`XFillRectangle`	Fills the specified rectangle or rectangles as if a four-point `FillPolygon` protocol request were specified for each

```
    window = XCreateSimpleWindow(display, RootWindow(display, s),
             10, 10, 200, 200, 1,
                   BlackPixel(display, s), WhitePixel(display,
                   s));

    /* select kind of events we are interested in */
    XSelectInput(display, window, ExposureMask | KeyPressMask);

    /* map (show) the window */
    XMapWindow(display, window);

    /* Start the Event-Request Loop*/
    for (;;)
    {
       XNextEvent(display, &event);

       /* draw or redraw the window */
       if (event.type == Expose)
       {
           XFillRectangle(display, window, DefaultGC(display, s),
           20, 20, 10, 10);
           XDrawString(display, window, DefaultGC(display, s), 50,
           50, msg, strlen(msg));
      }
      /* exit on key press */
       if (event.type == KeyPress)
            break;
    }

    /* Cleanup */

    XCloseDisplay(display);

    return 0;
}
```

22.4.4 Using XCB

In this section, we give basic examples of programming in XCB. Compare the complexity and extant documentation to the example programs we provide for Xlib.

22.4.4.1 Compiling an XCB Program

Be aware that, on a PC-BSD system, you will have to download and install the GNU C compiler for the example programs to run, if that has not already been done by the system administrator. You can do this for yourself by using the procedures in the AppCafe in PC-BSD or by using the repository in Solaris.

On our PC-BSD system, we used the following compiler command with the options and arguments shown. On Solaris, you can substitute `gcc` for `gcc47`:

```
gcc47 -Wall input_file.c -o output_file -lxcb
```

You may also use the following compiler command:

```
gcc47 -Wall input_file.c -o output_file 'pkg_config -cflags -libs
xcb'
```

22.4.4.2 Sample XCB Client Application Programs

Following are three elementary sample XCB programs. Each is preceded by a statement of what the program does.

22.4.4.2.1 XCB Example Program **xcb_simple.c** Objective: Place a simple window on screen. Launch it by typing **`xcb _ simple &`** and notice what happens.

Code:

```
#include <unistd.h> /* pause() */
#include <xcb/xcb.h>
int
main ()
{
  xcb_connection_t       *c;
  xcb_screen_t           *screen;
  xcb_window_t           win;

/* Initialization */

  /* Open the connection to the X server */
  c = xcb_connect (NULL, NULL);

  /* Get the first screen */
  screen = xcb_setup_roots_iterator (xcb_get_setup (c)).data;

  /* Ask for our window's Id */
  win = xcb_generate_id(c);

  /* Create the window */
  xcb_create_window (c,                /* Connection     */
    XCB_COPY_FROM_PARENT,              /* depth (same as root)*/
    win,                               /* window Id          */
    screen->root,                      /* parent window      */
    0, 0,                              /* x, y               */
    250, 250,                          /* width, height      */
    10,                                /* border_width       */
    XCB_WINDOW_CLASS_INPUT_OUTPUT,     /* class              */
    screen->root_visual,               /* visual             */
    0, NULL);                          /* masks, not used yet */
  /* Map the window on the screen */
  xcb_map_window (c, win);

/* Make sure commands are sent before we pause, so window is
shown */
  xcb_flush (c);

  pause ();    /* hold client */
/* Cleanup */

  xcb_disconnect(c);
  return 0;
}
```

EXERCISE 22.21

What basic component of an X Windows client application is missing from the preceding program, and particularly what aspect or part of that component? How do you close the window without using the KDE4 window manager kill button?

22.4.4.2.2 XCB Example Program **2ndxcbdraw.c** Objective: Draw a rectangular box in a window

Code:

```
#include <stdlib.h>
#include <stdio.h>
#include <xcb/xcb.h>
int
main ()
{
  xcb_connection_t          *c;
  xcb_screen_t              *screen;
  xcb_drawable_t            win;
  xcb_gcontext_t            foreground;
  xcb_generic_event_t       *e;
  uint32_t                  mask = 0;
  uint32_t                  values[2];

  /* geometric objects */

  xcb_rectangle_t      rectangles[] = {
    { 10, 50, 40, 20},
    { 80, 50, 10, 40}};

/* Initialization */

  /* Open the connection to the X server */
  c = xcb_connect (NULL, NULL);
  /* Get the first screen */
  screen = xcb_setup_roots_iterator (xcb_get_setup (c)).data;

  /* Create black (foreground) graphic context */
  win = screen->root;

  foreground = xcb_generate_id (c);
  mask = XCB_GC_FOREGROUND | XCB_GC_GRAPHICS_EXPOSURES;
  values[0] = screen->black_pixel;
  values[1] = 0;
  xcb_create_gc (c, foreground, win, mask, values);

  /* Ask for our window's Id */
  win = xcb_generate_id(c);

  /* Create the window */
  mask = XCB_CW_BACK_PIXEL | XCB_CW_EVENT_MASK;
  values[0] = screen->white_pixel;
  values[1] = XCB_EVENT_MASK_EXPOSURE;
  xcb_create_window (c,                          /* Connection       */
       XCB_COPY_FROM_PARENT,                     /* depth            */
       win,                                      /* window Id        */
       screen->root,                             /* parent window    */
       0, 0,                                     /* x, y             */
       150, 150,                                 /* width, height    */
       10,                                       /* border_width     */
       XCB_WINDOW_CLASS_INPUT_OUTPUT,            /* class            */
       screen->root_visual,                      /* visual           */
       mask, values);                            /* masks            */

  /* Map the window on the screen */
  xcb_map_window (c, win);

  /* Flush the request */
  xcb_flush (c);

/* Start the Event-Request loop */

  while ((e = xcb_wait_for_event (c))) {
    switch (e->response_type & ~0x80) {
    case XCB_EXPOSE: {

      /* Draw the rectangles */
      xcb_poly_rectangle (c, win, foreground, 2, rectangles);

      /* Flush the request */
      xcb_flush (c);

      break;
    }
    default: {
      /* Unknown event type, ignore it */
      break;
    }
    }
    /* Free the Generic Event */
    free (e);
  }

/* Cleanup */

  xcb_disconnect(c);
return 0;
}
```

22.4.4.2.3 XCB Example Program **xcb_example.c** Objective: Draw a rectangular box in a window

Code:

```
/* Simple XCB application drawing a box in a window */
/* to compile it use: gcc47 -Wall xcb_example.c -o xcb_example -lxcb
*/
#include <xcb/xcb.h>
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
xcb_connection_t      *c;
xcb_screen_t          *s;
xcb_window_t           w;
xcb_gcontext_t         g;
xcb_generic_event_t   *e;
uint32_t              mask;
uint32_t              values[2];
int                   done = 0;
xcb_rectangle_t       r = { 30, 30, 70, 70 };

/* Initialization*/

/* open connection with the server */
c = xcb_connect(NULL,NULL);
     if (xcb_connection_has_error(c)) {
     printf("Cannot open display\n");
     exit(1);
}
/* get the first screen */
s = xcb_setup_roots_iterator (xcb_get_setup(c)).data;
/* create black graphics context */
g = xcb_generate_id(c);
w = s->root;
mask = XCB_GC_FOREGROUND | XCB_GC_GRAPHICS_EXPOSURES;
values[0] = s->black_pixel;
values[1] = 0;
xcb_create_gc(c, g, w, mask, values);

/* create window */
w = xcb_generate_id(c);
mask = XCB_CW_BACK_PIXEL | XCB_CW_EVENT_MASK;
values[0] = s->white_pixel;
values[1] = XCB_EVENT_MASK_EXPOSURE | XCB_EVENT_MASK_KEY_PRESS;
xcb_create_window(c, s->root_depth, w, s->root,
                                 10, 10, 150, 150, 1,
                                 XCB_WINDOW_CLASS_INPUT_
OUTPUT, s->root_visual,
                                 mask, values);

/* map (show) the window */
xcb_map_window(c, w);

xcb_flush(c);

/* Start the Event-Request loop */

while (!done && (e = xcb_wait_for_event(c))) {
     switch (e->response_type & ~0x80) {
     case XCB_EXPOSE: /* draw or redraw the window */
          xcb_poly_fill_rectangle(c, w, g, 1, &r);
          xcb_flush(c);
          break;
     case XCB_KEY_PRESS: /* exit on key press */
          done = 1;
          break;
     }
     free(e);
     }
     /* Cleanup */
     xcb_disconnect(c);

     return 0;
}
```

22.4.4.3 X Events in XCB

The following is a simple example of how to monitor XCB events of the types shown.

22.4.4.3.1 XCB Example Program **xcb_events.c** Objective: Report screen coordinates of three-button mouse presses, and movement of the current position in the window created

Code:

```
#include <stdlib.h>
#include <stdio.h>
#include <xcb/xcb.h>
void
print_modifiers (uint32_t mask)
{
  const char **mod, *mods[] = {
    "Shift", "Lock", "Ctrl", "Alt",
    "Mod2", "Mod3", "Mod4", "Mod5",
    "Button1", "Button2", "Button3", "Button4", "Button5"
  };
  printf ("Modifier mask: ");
  for (mod = mods; mask; mask >>= 1, mod++)
    if (mask & 1)
      printf(*mod);
  putchar ('\n');
}

int
main ()
{
  xcb_connection_t    *c;
  xcb_screen_t        *screen;
  xcb_window_t         win;
  xcb_generic_event_t *e;
  uint32_t             mask = 0;
  uint32_t             values[2];

  /* Open the connection to the X server */
  c = xcb_connect (NULL, NULL);

  /* Get the first screen */
  screen = xcb_setup_roots_iterator (xcb_get_setup (c)).data;

  /* Ask for our window's Id */
  win = xcb_generate_id (c);

  /* Create the window */
  mask = XCB_CW_BACK_PIXEL | XCB_CW_EVENT_MASK;
  values[0] = screen->white_pixel;
  values[1] = XCB_EVENT_MASK_EXPOSURE       | XCB_EVENT_MASK_
BUTTON_PRESS   |
              XCB_EVENT_MASK_BUTTON_RELEASE | XCB_EVENT_MASK_
POINTER_MOTION |
              XCB_EVENT_MASK_ENTER_WINDOW   | XCB_EVENT_MASK_
LEAVE_WINDOW   |
              XCB_EVENT_MASK_KEY_PRESS      | XCB_EVENT_MASK_KEY_
RELEASE;
  xcb_create_window (c,                  /* Connection          */
      0,                                 /* depth               */
      win,                               /* window Id           */
      screen->root,                      /* parent window       */
      0, 0,                              /* x, y                */
      150, 150,                          /* width, height       */
      10,                                /* border_width        */
      XCB_WINDOW_CLASS_INPUT_OUTPUT,/* class               */
      screen->root_visual,               /* visual              */
      mask, values);                     /* masks               */

  /* Map the window on the screen */
  xcb_map_window (c, win);

  xcb_flush (c);

  while ((e = xcb_wait_for_event (c))) {
    switch (e->response_type & ~0x80) {
    case XCB_EXPOSE: {
      xcb_expose_event_t *ev = (xcb_expose_event_t *)e;

      printf ("Window %ld exposed. Region to be redrawn at
      location (%d,%d), with dimension (%d,%d)\n",
              ev->window, ev->x, ev->y, ev->width, ev->height);
      break;
    }
    case XCB_BUTTON_PRESS: {
      xcb_button_press_event_t *ev = (xcb_button_press_event_t *)e;
      print_modifiers(ev->state);

      switch (ev->detail) {
      case 4:
        printf ("Wheel Button up in window %ld, at coordinates
        (%d,%d)\n",
                ev->event, ev->event_x, ev->event_y);
        break;
      case 5:
        printf ("Wheel Button down in window %ld, at coordinates
        (%d,%d)\n",
                ev->event, ev->event_x, ev->event_y);
        break;
      default:
        printf ("Button %d pressed in window %ld, at coordinates
        (%d,%d)\n",
                ev->detail, ev->event, ev->event_x, ev->event_y);
      }
      break;
    }
    case XCB_BUTTON_RELEASE: {
      xcb_button_release_event_t *ev = (xcb_button_release_event_t
      *)e;
      print_modifiers(ev->state);

      printf ("Button %d released in window %ld, at coordinates
      (%d,%d)\n",
              ev->detail, ev->event, ev->event_x, ev->event_y);
      break;
    }

    default:
      /* Unknown event type, ignore it */
      printf("Unknown event: %d\n", e->response_type);
      break;
    }
    /* Free the Generic Event */
    free (e);
  }

  return 0;
}
```

22.4.5 Using the Qt Toolkit

As seen in Figure 22.21, there are basically two ways an X Windows client application program can use its program code to work in conjunction with the X protocol running on the X server: directly using XCB or Xlib library calls, or by using a toolkit specifically designed to act as a simple-to-use intermediary that minimizes the coding complexity of dealing with X protocol structure and functions. Qt is just such an intermediary toolkit. It is a library of C++ code that allows you to create a client application program without dealing with the details of XCB, Xlib, or the X protocol.

In the following exercises, we give you an overview of Qt programming as implemented in a CUI terminal or console window. It is also possible to use a GUI-based IDE tool, such as Qt Designer or the more complete Qt Creator to expedite a project that involves the harnessing of a Qt-generated GUI to a client application program's data generation code.

22.4.5.1 Creating an Executable Qt Program

The Qt libraries are installed by default on a PC-BSD system. Qt has its own compiling, linking, and assembling procedure, as shown. The following steps show how to use the Qt procedure to compile a Qt program, create a Qt project, and execute a client application program on our base PC-BSD system.

You do not use the GNU C++ compiler to do any of these operations.

Step 0: Create an empty directory under your home directory, with a name like **qtprogs1**. Make that directory the present working directory. Use a text editor of your choice to enter and save the Qt code of any of the following example exercises into a file with the file extension **.cpp**—for example, **exercise1.cpp**, which contains Qt client application program code.

Step 1: At the shell prompt, type **`qmake-qt4 -project`**.

Step 2: At the shell prompt, type **`qmake-qt4`**.

Step 3: At the shell prompt, type **`make`**.

Step 4. In the directory you will now have five files: **exercise1.cpp**, **Makefile**, **exercise1.o**, **qtprogs1.pro**, and **qtprogs1**.

Step 5: At the shell prompt, type **`./qtprogs1`**.

Step 6: The graphics contained in the Qt program you entered into **exercise1.cpp** are now shown on the screen. Use the kill window button to close this Qt window.

22.4.5.2 Qt Examples

Here is the simplest Qt client application program you can enter. Notice how much shorter it is than the XCB or Xlib code shown in the preceding sections. Follow the instruction steps in Section 22.4.5.1 to create the program and execute it.

Qt EXERCISE 22.1: SIMPLEST Qt PROGRAM

```
// helloWorld/main.cpp
#include <QApplication>
#include <QLabel>
int main(int argc, char *argv[])
{
         QApplication a(argc, argv);
         QLabel label("Hello World");
         label.show();
         return a.exec();
}
```

Qt EXERCISE 22.2: SIMPLE Qt PROGRAM WITH A GUI

In this example, we create and show a built-in Qt text edit capability in an open window. This represents a simple Qt program that has a GUI. The line numbers shown to the left of the code should be omitted, they are only there for reference in the dialog of explanation that follows.

Here is the code:

```
1        #include <QApplication>
2        #include <QTextEdit>
3
```

```
 4        int main(int argv, char **args)
 5        {
 6            QApplication app(argv, args);
 7
 8            QTextEdit textEdit;
 9            textEdit.show();
10
11            return app.exec();
12        }
```

Dialog of explanation and description:

Lines 1 and 2 include the header files for `QApplication` and `QTextEdit`, which are the two classes that Qt uses. Qt is an object-oriented programming language that uses C++ class and object descriptions and functionality. All Qt classes have a header file named after them.

Line 4 declares the variables and opens the program.

Line 6 creates a `QApplication` object. This object manages application-wide resources and is necessary to run any Qt program that has a GUI. It needs `argv` and `args` because Qt accepts a few command line arguments for this object.

Line 8 creates a `QTextEdit` object. A *text edit* is a visual element in the GUI. In Qt, we call such elements *widgets*, short for *window gadgets*. Examples of other Qt widgets are scroll bars, labels, spin boxes, sliders, and radio buttons. A widget can also be a container for other widgets, a dialog area, or a main application window.

Line 9 shows the text edit on the screen in its own window frame. Since widgets also function as containers (for instance `QMainWindow`, which has toolbars, menus, a status bar, and a few other widgets), it is possible to show a single widget in its own window. Widgets are not visible by default; the function `show()` makes the widget visible.

Line 11 makes the `QApplication` object enter its event loop, similar to XCB and Xlib client application programs. When a Qt client application is running, events are generated and sent to the widgets of the application. Examples of events, as seen in XCB and Xlib, are mouse button presses, mouse cursor movements, and key strokes pressed on the keyboard. When you type text in the text edit widget, it receives key press events and responds by drawing the text that was typed.

Qt EXERCISE 22.3: ADDING A QUIT BUTTON

In a real application, you would usually create more than one widget to allow a rich and varied dialog between client application data-generating code and Qt code. We will now show a simple example of a `QPushButton` beneath the text edit window created in Qt

Exercise 22.2. The button will exit the `QTextEdit` application when pushed (i.e., clicked on with the mouse). Again, disregard the line numbers shown, since they are only used to reference the code in the dialog that follows the code.

Here is the code.

```
 1      #include <QtGui>
 2
 3      int main(int argv, char **args)
 4      {
 5          QApplication app(argv, args);
 6
 7          QTextEdit *textEdit = new QTextEdit;
 8          QPushButton *quitButton = new QPushButton("&Quit");
 9
10          QObject::connect(quitButton, SIGNAL(clicked()), qApp,
            SLOT(quit()));
11
12          QVBoxLayout *layout = new QVBoxLayout;
13          layout->addWidget(textEdit);
14          layout->addWidget(quitButton);
15
16          QWidget window;
17          window.setLayout(layout);
18
19          window.show();
20
21          return app.exec();
22      }
```

Dialog of explanation and description:

Line 1 includes `QtGui`, which contains all of Qt's GUI classes.

Lines 7 and 8 create two pointer objects to be used to reference the classes of objects below.

The next line illustrates probably the most important Qt call.

Line 10 uses Qt's s*ignals and slots* mechanism to make the application exit when the `Quit` button is pushed. A *slot* is a function that can be invoked at run time using its name (as a literal string). A *signal* is a function that when called will invoke slots registered with it; we call that to connect the slot to the signal and to emit the signal. So, `quit()` is a *slot* of `QApplication` that exits the application; `clicked()` is a signal that `QPushButton` emits when it is pushed.

As a programming reminder for C++, :: is called the (binary) *scope resolution operator.* By using the scope resolution operator, you can address member functions outside of a class. Also remember that the scope resolution operator specifies that the identifier which is on the right belongs to the data type or class on the left.

The static `QObject::connect()` function takes care of connecting the slot to the signal. `SIGNAL()` and `SLOT()` are two macros that take the function signatures of the signal and slot to connect. We also need to give pointers to the objects that should send and receive the signal.

Line 12 creates a `QVBoxLayout`. As mentioned, widgets can contain other widgets. It is possible to set the bounds (the location and size) of child widgets directly, but it is usually easier to use a layout. A layout manages the bounds of a widget's children. `QVBoxLayout` places the children in a vertical row.

Line 13 and 14 adds the text edit and button to the layout.

Line 17 sets the layout on a widget.

Line 19 uncovers the window.

Line 21 starts the event loop.

Qt EXERCISE 22.4: CONNECTING SIGNALS AND SLOTS

The following Qt code places three widgets in a window, and defines interconnections between the signal elements and slot elements of those widgets.

```
// signalSlot2/main.cpp
#include <QApplication>
#include <QVBoxLayout>
#include <QLabel>
#include <QSpinBox>
#include <QSlider>
int main(int argc, char *argv[])
{
         QApplication a(argc, argv);
         QWidget window;
         QVBoxLayout* mainLayout = new QVBoxLayout(&window);
         QLabel* label = new QLabel("0");
         QSpinBox* spinBox = new QSpinBox;
         QSlider* slider = new QSlider(Qt::Horizontal);
         mainLayout->addWidget(label);
         mainLayout->addWidget(spinBox);
         mainLayout->addWidget(slider);
         QObject::connect(spinBox, SIGNAL(valueChanged(int)),
         label, SLOT(setNum(int)));
         QObject::connect(spinBox, SIGNAL(valueChanged(int)),
         slider, SLOT(setValue(int)));
         QObject::connect(slider, SIGNAL(valueChanged(int)),
         label, SLOT(setNum(int)));
         QObject::connect(slider, SIGNAL(valueChanged(int)),
         spinBox, SLOT(setValue(int)));
         window.show();
         return a.exec();
}
```

SUMMARY

The operability of UNIX is greatly improved from the user perspective by deployment of a *graphical user interface* (GUI). A common GUI system is built upon a network protocol called the X Window System. This GUI system can be classified as either *integrated* or *non-integrated*. A nonintegrated system generally utilizes only the functionality of a window manager. An integrated system generally couples the window manager with other higher-level programs that achieve *desktop management* and *session management*. An example of an integrated system is KDE4.

The X Window System is a network protocol and contains device-specific drivers for Intel-based PC hardware. The X Window System is used for networked graphical interaction between a user and one or more computer systems running UNIX. The chief arbiter of the interactive dialog between user and computer system is the *window manager*. The FVWM window manager offers all of the amenities of other popular window systems, and additionally allows you to manage the graphical output from UNIX application programs. The user interface has two basic parts: the *application user interface* (AUI), which is how each client application presents itself in one or more windows on the server screen display, and the window manager or *management interface*, which controls the display of and organizes all client windows.

The basic model of interactivity in the X Window System is an *event–request* loop between the application *client* and the graphical *server*. With applications written for the X Window System, the client application can process input events, do the work necessary to form a response to the events, and then output the responses as requests for graphical output to the server.

The X Window System, and FVWM in particular, are highly customizable to suit the interactive needs of a wide range of users. In this chapter we covered two approaches to changing the appearance and functionality of a nonintegrated window system, and the window manager as well. The first approach involved changing the characteristics of applications that run under the X Window System by specifying command line options. The second approach involved modifying a predefined sample initialization file, named **~.fvwm/config**, for the window manager FVWM, and then invoking that initialization file.

We covered the functionality of the predominant open-source integrated desktop management system KDE4. We showed how this system can be used to expedite your work within the UNIX environment, particularly with regard to personal productivity and file management operations. We specifically showed the customization possible within this system to allow a user to work more efficiently.

We showed several elementary sample client application programs for the X Window System, coded to call upon three standard and very common toolkit libraries, Xlib, XCB, and Qt. We stressed the two most important aspects of client application program creation for the beginner:

- A client application program has two parts: a data generation part that uses code in C, C++, or another high-level programming language to produce the numbers, text, files, and data structures; and a user interface (UI) part which produces the actual graphics that display the data generation part.
- A client application program is made up of initialization, event–request loop, and cleanup sections.

We also showed many detailed examples of the toolkit code itself and what the code accomplished in the context of the sample programs.

QUESTIONS AND PROBLEMS

1. Give definitions, in your own words, for the following terms as they relate to the X Window System: *window system*, *window manager*, *desktop manager*, *client*, *server*, *focus*, *iconify*, *maximize*, *minimize*, *xterm*, *application user interface*, *management interface*.
2. Which X Window System window manager is used on your computer system? How can you identify and recognize which window manager you are using by default?
3. Make the following changes to your **~/.fvwm/config** file to set `DeskTopSize` to 3×3. What effect does that have when you restart FVWM?

   Then, use the **Root menu>Fvwm Simple Config Ops** fly-out menu to change the focus of **Click to Focus**. What effect does that have?
4. Which command allows another user to have their windows displayed on your screen under the X Window System? What would be the advantages of doing this? What would be the disadvantages of doing this? Explain why this is even possible at all under the X Window System.
5. Identify the xterm options that are set on your computer system. What is the default size of an xterm window? What is the default background color for an xterm window? What do you think are the most useful xterm options for you?
6. Use the AppCafe on a PC-BSD system to download and install another window manager, such as Awesome or FVWM-Crystal. Then login and invoke that new window manager and compare its operability, functionality, and ease of use to FVWM. Which window manager do you like best and why?
7. a. When you hold down the left-most mouse button when the screen cursor is in the root window of your X Window System display, what appears on your screen?

What appears when you hold down the middle mouse button? What appears when you hold down the right-most mouse button?

b. What controls the appearance and content of the menus that are presented to you when you take these actions?

c. If you hold down the middle mouse button when the screen cursor is over the title bar of a window on your display, what happens? What is presented to you?

8. Do all windows launched on your X Window System display have the same components—that is, scroll bars, iconify button, title bar, and resize handles? What facility controls the look and feel of these components? How do these components compare in function and operation to what you might be familiar with from another GUI—for example, when using OS X or Windows?

9. Use your favorite Web browser to explore the site www.X.org. What are the objectives of this organization? What is another good source of information on the X Window System?

10. Use Gimp to design a bitmapped image for use as an icon in a pull-down menu. For example, if you were going to design a menu choice for reading from a file, your bitmapped image might look like a book that is open for reading. Save the image, and then use an image-viewing application to view the image you designed.

11. After completing Problem 10, find an X-based application on your network that allows you to customize menu items. Then, design icon images for use with the application using Gimp and install them for use with the application.

12. For KDE4, compare the file maintenance facilities available through Dolphin with the UNIX commands that do file maintenance. What are the advantages of the desktop manager's file facilities? What are the advantages of the UNIX commands that do file maintenance, particularly ZFS commands and Disk Manager in PC-BSD? Can you see the advantage of using both at the same time?

13. What is a session manager, and how is it different from a desktop management system or a window manager?

14. What are the three major components of KDE4? Give a brief description of each one of them.

15. How can you add an application icon and menu choice to the Kickoff Application Launcher application menus?

16. Outline the installation procedure for the KDE4 system if you obtain the software as a package over the Internet. When would it be necessary for you to do this installation?

17. How can you upgrade KDE4 components? What components of KDE4 would you add or upgrade? From what repository are these components available?

18. Why would someone want to do a nonintegrated installation of UNIX—that is, either without a GUI (with only a text-based interface to the system) or running only a bare-bones window manager like FVWM?
19. Why are server-class installations of UNIX done without a GUI?
20. The primary task, and biggest challenge, of programming a client application in the X Window System is connecting the output of code that generates or actually is data—such as numbers, text strings, files, file structures, and so on—to a user UI implemented by one of the toolkits we show in this chapter. Of course, if the only objective of the UI of a client application program is to produce output graphics, then it is advisable to partition the client application into a data generation part (if there is one) and a graphics production part. Separating these two parts out is helpful and useful when the code for each needs to be modified, maintained in the future, or documented and understood by other developers or team programming members, and the connectivity between the two discrete parts must remain the same.

    Examine all of the program examples we provide for Xlib, XCB, and Qt, and make a brief list of where the data in each program is generated, so that each of the toolkits can make graphics out of it. Also describe how that data is passed to the toolkit that uses the data, either as literal arguments, or as data structure mechanisms.

For example, since Xlib uses mostly a procedural paradigm to pass the data-generating components to the UI components, an entry in your answer tableau for the program example **Xlib test1.c** should appear like this:

```
Xlib test1.c  XFillRectangle(…,20,20,10,10),   Integers, string
              XDrawString(…, msg, …)
```

Main Page Table of Contents for FVWM

1. Name
2. Synopsis
3. Description
4. Options
5. Anatomy of a Window
6. The Virtual Desktop
7. Use on Multi-Screen Displays
8. Xinerama Support
9. Initialization

31. List of Fvwm Commands
    - 31.1. Menus
    - 31.2. Miscellaneous Commands
    - 31.3. Window Movement and Placement
    - 31.4. Focus & Mouse Movement
    - 31.5. Window State
    - 31.6. Mouse, Key & Stroke Bindings
    - 31.7. Controlling Window Styles
    - 31.8. Window Styles
    - 31.9. Controlling the Virtual Desktop
    - 31.10. User Functions and Shell Commands
    - 31.11. Conditional Commands
    - 31.12. Module Commands
    - 31.13. Session Management Commands
    - 31.14. Colorsets
32. Environment
33. Authors
34. Copyright
35. Bugs

CHAPTER 23

UNIX System Administration Fundamentals

Objectives

- To illustrate the most efficient way to do a fresh install of a 64-bit, X86 architecture version from an ISO-created DVD medium, using a GUI installer onto a single hard disk system, with a KDE4 (PC-BSD) or GNOME (Solaris) GUI desktop
- To suggest system build design philosophy.
- To describe how to gracefully bring the system down
- To illustrate how to add additional users and groups to the system
- To show how to design and maintain user accounts
- To show post installation hardware, such as disk drives
- To provide strategies for backup and archival of system files and user files
- To show how to upgrade and maintain the operating system software
- To show how to add/update/remove user application package repository software
- To show how to monitor the performance of the system, particularly with system log files
- To provide strategies for system security, particularly ACLs, and system firewall
- To cover the commands and primitives (PC-BSD shown; Solaris only in parentheses)

```
adduser, appcafe, beadm, cpio, (dispadmin), dmesg, dump,
(format), getfacl, gpart, gparted, (groupadd), (groupdel),
gtar, ifconfig, (ipadm), ipfw, netstat, (ndd), pf,
pc-fwmanager, pc-systemupdatertray, pc-updategui,
pc-updatemanager, pc-zmanager, pfctl, pkg, restore,
(rmformat), rmuser, rsync, setfacl, shutdown, su, sudo,
(svcadm), (svcs), sysctl, tar, (ufsdump), (ufsrestore),
uncompress, (useradd), (userdel), (usermgr), (usermod), zfs,
zpool
```

23.1 INTRODUCTION

In order to install and maintain a UNIX system composed of both hardware and software components, it is necessary to perform the set of common tasks shown in the following sections. These common tasks may be performed by an individual exclusively for their own use on their own personal desktop/laptop/tablet computer, or may be performed by an appointed administrator for a more complex computer system used by many people. It is also possible to divide these common tasks into those performed by an autonomous administrator and a group of ordinary users. In this chapter, we use a "learning by doing" approach aimed at the individual rather than at an appointed system administrator. Even though we show the basics of those common tasks, it is possible to extrapolate from what is presented to the wider context of a larger computer system.

In order to do any of the system administration tasks in this chapter, it is necessary to either have superuser or root user privileges on the system. That means you need to know the superuser password, which you (or a designated system administrator) can establish at installation for PC-BSD and Solaris.

Both the `su` and `sudo` commands are used to execute programs and other commands with root permissions. The root user has maximum permissions, and can do anything to the system that a system administrator needs to. Normal users execute programs and commands with reduced permissions.

To execute something that requires maximum permissions, you will first have to execute the `su` or `sudo` commands.

Using the `su` command makes you the superuser—or root user—when you execute it with no additional options. You are prompted to enter the root account's password. Also, the `su` command allows you to switch to any user account. If you execute the command `su bob`, you'll be prompted to enter **bob**'s password, and the command shell current working directory will switch to **bob**'s home directory.

Once you're done running commands in the root shell, you should type `exit` to leave the root shell and go back to limited-privileges mode. The root shell is not the login shell.

In comparison, `sudo` runs a single command with root privileges. When you execute `sudo command`, the system prompts you for the current user account's password before running `command` as the root user. More information on the `sudo` command can be found in Section 23.9.2.1.

In our "learning by doing" approach, the common tasks are

1. Doing a fresh install of a 64-bit, X86 architecture version from an ISO-created DVD medium, using a GUI installer, onto a single hard disk system, with a KDE4 or GNOME GUI desktop; doing a preliminary configuration of that system
2. Gracefully bringing the system down
3. Adding additional users and groups to the system, and show how to design and maintain user accounts
4. Adding hardware to the system, such as disk drives
5. Providing strategies using the traditional and generic UNIX commands, to backup and archive the system files and user files
6. Upgrading and maintaining the operating system, and adding/updating/removing user application package repository software to both increase functionality and update existing packages
7. Monitoring the performance of the system and tune it for optimal performance characteristics
8. Providing strategies for system security to harden the individual desktop computer
9. Providing network connectivity strategies, both on a LAN and the Internet

We show how to do these nine common tasks both in PC-BSD and Solaris, the representative systems of the two major families of UNIX.

Some examples of advanced extensions on these basic tasks, and which we do not cover, might be listed as

1. Installing from a user-created, bootable USB thumb drive, running a "live" version from DVD or a persistent USB thumb drive, or installing over a LAN or from a server
2. Doing an advanced, text-based installation of a server with a complex configuration
3. Maintaining a large user base across multiple machines and networks, using ACLs
4. Configuring higher levels of RAIDZ or multipartitioned disks and volumes with multiple operating systems on them
5. Using commercial build or backup software, such as Norton Ghost
6. Hot swapping hard drives and software
7. Hand building software systems from source or using advanced package repository resources
8. Using the UNIX system exclusively to stream media via NAS to multiple displays
9. Doing the common system administration tasks on OS X, iOS, Linux, or Android

10. Writing and administering malware-fighting programs
11. Integrating the system with cloud storage and computing, or implementing a full UNIX, Apache, MySQL, Python (UAMP) stack

To extend some of the system administration topics covered in this book so that they are reflective of modern UNIX systems, we provide two additional chapters as follows: Chapter 24 on ZFS details how a modern UNIX file system can be utilized to efficiently manage system and user files; Chapter 25 on virtualization methodologies shows PC-BSD Jails, Solaris 11 Zones, and VirtualBox virtual machines (VMs). These two additional chapters show how to implement further levels of system security, as well as to expedite the system administration techniques and methods shown in this chapter.

23.2 DOING A FRESH INSTALL FROM ISO-CREATED DVD MEDIA AND PRELIMINARY SYSTEM CONFIGURATION

This section assumes that you have already obtained and burned to disk the ISO file of the DVD media for PC-BSD or Solaris, and that your system is a 64-bit, X86 architecture computer with at least one entire hard disk that you are willing to install the UNIX system on. If you have not obtained a DVD or do not have a 64-bit computer, you should follow the online documentation at www.pcbsd.org or www.oracle.com to get the ISO image of the DVD media, or use an earlier release of either of the software systems that are designed for 32-bit machines. "Obtaining the DVD media" means downloading an ISO file for the release of the software you want to use, and then burning that ISO image to a DVD. We do not give instructions here for that process. Be aware that some of the systems administration tasks illustrated for PC-BSD and Solaris are done in a different way in earlier releases of the software!

This section also assumes that you will not be permanently running the "live" version of either of the UNIX systems from persistent or nonpersistent media, like a USB drive or CD/DVD.

A few words about ISO files and media at this point would be instructive. We have chosen to do our installs in this chapter from DVDs created by burning ISO files downloaded from both www.pcbsd.org and www.oracle.com. That is because the vast majority of desktop, server, and laptop computers being sold today, and that have been sold for the past 15 years, come equipped with a DVD drive as standard. This may change in the future, where the predominant form of removable media hardware may become something else, such as USB 3.0 thumbdrives, for example. It is presently possible to use a downloaded ISO file for the installation of the operating systems placed on a USB external medium, but we chose not to use this method. If you want to use this method, you can if it fits your use case, hardware requirements, and data storage model, and most importantly, is convenient and easy for you.

Also, as shown in Chapter 25, it is possible to use the downloaded ISO file without burning it to DVD media to install the operating system into a VM environment. And you can use the downloaded ISO file in this chapter as well to directly install the operating system without using DVD media. Again, the method of installation you choose is your personal preference, based upon what works best for you on your particular computer hardware, what your use case is, and how you store your data.

If you want to install either PC-BSD or Solaris in a virtual environment, so that it runs simultaneously with some other operating system on your computer's hardware, we give specific instructions on how to do this in Chapter 25, "Virtualization Methodologies." In Chapter 25, Section 4.5, we give the specifics of installing PC-BSD and Solaris as "guest" operating systems on Ubuntu Linux and Windows hosts, using VirtualBox.

23.2.1 Preinstallation Considerations

Each of the UNIX systems we will show an installation of, PC-BSD and Solaris, have minimum hardware and driver requirements that can be determined before you begin an install. These hardware requirements are listed in the documentation online for each of the systems. Once you have determined that your computer system hardware meets or exceeds those minimum requirements, the following questions suggest some other important considerations you can make over and above the default installation choices before proceeding with the installation:

1. How many hard disks does your computer have, and how are you planning to use the Zettabyte File System (ZFS) on those disks? For example, if you only have one hard disk drive, that will be the bootable system disk and the file system data disk. If you have two or more hard disk drives, will you *mirror* the bootable system disk and all other data from the main drive on the other drives? In PC-BSD, mirroring on two or more disks can be accomplished at system installation.

   Most importantly, if your hardware can support multiple hard drives, we recommend that you install the operating system on one hard drive (preferably a solid state drive [SSD]), and all of the user data on another single disk or an array of hard disk drives. That way, if the operating system and its bootable hard disk drive become corrupted or unusable for some reason, your user data is on a separate hard disk or array of hard disks. As shown in Section 23.7.3.1, which deals with the most practical methods of operating system upgrades, you can then simply replace the operating system hard disk and reinstall either the current version of the operating system or a newer version. You can then use the ZFS facilities to reattach the data hard drives to the new operating system and its hard disk drive. This is highly valuable not only for desktop computers but also server-class systems as well. The way that your data is deployed on your disks is a critical design consideration when you are building your system, and is highly dependent on the particular use case that is guiding it.

2. Do you have a wireless connection to a LAN and the Internet? A wired connection through DHCP is automatically done in both systems during the default install.

3. How many users are you going to initially establish at installation, and what are their user profiles going to be? For example, what users will have administrative privileges other than yourself, and what kind of security will each profile have? Also, what user

groups are you going to establish at installation, and how are you going to manage groups in a post installation environment?

4. Who will be responsible for the nine systems administration tasks listed previously? For example, this will influence your disk management tasks concerning file systems for users and projects, according to the data storage model you employ.
5. What kind of software tools do you want to include for the kinds of tasks you and your user base will be doing? For example, during installation, you are able to add packages on top of the default package installation to help accomplish those tasks. How are the user groups established in item 3 going to have access privileges to this software? What are your policies with respect to group access to software tools, and how do you enforce this policy?
6. What kind of GUI windowing system do you want to install with the system? You have a choice at installation of KDE4, Gnome, and others, and your previous experience and preferences with a particular style of desktop environment can be implemented at installation. Will you be doing a server install, based on your use case? A server install, and the management of a server system, involves another whole universe of considerations and design decisions.

EXERCISE 23.1

Make a detailed listing on paper of your answers to these six questions before you begin to install your operating system. Then, read through the following sections showing the procedures for actual installation, and for each of your answers, determine ahead of time how you will proceed. This exercise is meant to serve as a "dry run" through any particular path you might take through the installation process.

23.2.2 GUI Install of PC-BSD

The most efficient, and patented, way to install PC-BSD on your computer is to follow the installation instructions in the PC-BSD handbook for the release of the software you want to install. Practically speaking, you can proceed through the installation from start to finish in a very intuitive manner, without using the handbook. Then, once the system is installed, you can consult the handbook, which is opened via clicking on an icon on the KDE4 desktop, for further detailed descriptions of the GUI installation process. In addition, versions of the PC-BSD handbook are available online for earlier releases of the software, and the general process of doing a GUI installation has not changed considerably since the earlier releases. This may change somewhat in later releases of the software, but certainly will not detrimentally affect using a purely intuitive initial installation scenario.

EXERCISE 23.2

Why would it be dangerous to do a full disk install of a new installation on a disk that has legacy archived user data on it?

23.2.3 Postinstall Configuration

To do an efficient and expeditious postinstallation of PC-BSD, refer to the instructions in the PC-BSD handbook for the version of the software you are interested in.

23.2.4 GUI Install of Solaris

In order to permanently install Solaris on your computer system, you must take the following steps:

1. Boot to the Live Media ISO-created DVD. After booting to the DVD, the Live Media GNU Grand Unified Bootloader (GRUB) menu will appear briefly. The default boot environment (first option listed) will be used if you do not interrupt the GRUB menu and choose an alternate boot environment. Use the default.
2. The first screen allows you to select the keyboard layout that you will be using. The information at the top of the screen describes the version of Solaris being installed. Press **`<Enter>`** to choose the default of US English.
3. The next screen prompts you to enter the language that you wish to use. Press **`<Enter>`** to choose English.
4. The Live Media installer will boot the system, configure devices, and launch the desktop GUI.

   When the command line login prompt appears, wait a few minutes for the desktop GUI to start up.

   The system is booted from the DVD and you can start using Solaris; however, Solaris is running in RAM and is not installed on the local disk. While booted from Live Media, any changes made will be lost when the system is rebooted.
5. There are four icons on the desktop. Click on the Device Driver Utility to view the status of the devices on your system. If you are prompted for a password, enter **`solaris`**. It will take a few minutes for the system to gather the device information before it will display the Device Driver Utility.

   The Device Driver Utility will report any device issues. It is important to resolve any device issues of importance before continuing with the installation. If the system reports zero driver problems, or the device driver issues are not important for installation, click on the **`Close`** button to close the Device Driver Utility.

   The Device Driver Utility is only available on the x86 platform and will indicate which Solaris driver supports the various x86 components. If the utility detects a device that does not have a driver attached, that device is selected on the device list. You can choose to display more information about the device and install the missing driver. You can specify a Solaris IPS package or a file/URL path to the driver.

6. Begin the installation by clicking on the **`Install Solaris`** icon. The `Welcome` screen will open.

   The list on the left side of the window lists the following steps that you will follow to configure the system before installing the OS:

   a. Configure the root disk where the OS will be installed (sometimes called the boot disk)

   b. Set the time zone

   c. Create a user login account

   d. Begin the installation

7. Click on the **`Next`** button and the Disk Discovery window will appear. Select whether you will be installing the OS onto a local disk (internal or external disk) or an Internet Small Computer System Interface (iSCSI) disk accessible over the network using the iSCSI protocol. Select **`Local Disks`**.

8. Click on the **`Next`** button and the Disk Selection window will appear. You will be using the whole disk for the Solaris OS, so make that selection and click **`Next`**.

   Note: This procedure will erase everything on the disk.

9. The `Time Zone` screen appears. Select the correct region, location, time zone, date, and time, then click **`Next`**.

10. The `Users` screen will appear. On this screen, you will create a user account and enter a computer name for this system. When installing the OS using the Live Media installer, you must create a user account. This will be your login account. After the installation, you will not be allowed to log in directly as root.

    When creating a user account, follow these rules:

    - The username cannot be root.
    - The username must start with a letter.
    - The username may contain alphabetical characters, numbers, underscores (_), periods (**.**), or hyphens (**-**).

    After the installation using Live Media is complete and the system has been rebooted, the default behavior is to not allow root to log in from the login screen. You will first log in using the user account that was created during installation, and then, if desired, switch to the root account when root privileges are required. See Chapter 2, Figures 2.2 and 2.3, to see the appearance of the login screen dialog boxes.

Enter your own computer name or accept the default, **solaris**. The computer name, also called the *hostname*, cannot be blank.

11. Click **`Next`** and the `Support Registration` screen will be displayed.
12. Click **`Next`** and the installation summary screen will be displayed.
13. After reviewing the contents of the installation summary screen, click the **`Install`** button and the installation will begin. When the installation is complete, the `Finished` screen will be displayed.
14. Click the **`Reboot`** button to reboot the system. Be sure to remove the DVD so that the system boots from the internal disk and not the DVD. After rebooting, the system will display the GUI login screen.

    If the GRUB menu appears, the system is still booting from the installation media. Remove the installation media and make sure that the system is set up to boot from the local boot disk.
15. Logging in: By default, Solaris 11 does not allow the root user to log in at the login screen. You will first log in using the user account that was created during the installation. See Chapter 2, Figures 2.2 and 2.3, to see the appearance of the login screen dialog boxes.

    After you've logged in, use the `su` command to switch to the root account when root privileges are required. The first time that you assume the root role, you will authenticate using the password that you specified for the user account during the installation. At that point, you will receive a message that the password has expired, and you will be required to set a new password for the root account, as shown in the following code:

```
# su root
Password: xxx
Su: Password for user 'root' has expired
New Password: yyy
Re-enter new Password: yyy
su: password successfully changed for root
#
```

23.2.5 System Services Administration, Booting and Shutdown Procedures

This section details the general procedures for starting up, and gracefully shutting down PC-BSD and Solaris. It gives a brief overview of the steps those systems go through in successfully shutting down. It then outlines some of the important system services administration utilities available to the system administrator, most prominently SMF for Solaris. Additionally, we give examples of commands for manipulating and changing those system services by enabling and running new services.

23.2.5.1 The Boot Process

Basically, a UNIX system is booted in this sequence:

- The firmware on the main system board finds the bootable media.
- The bootloader program loads the monolithic kernel from the boot media into RAM and starts it.
- The kernel uses the commands passed by the bootloader to find the main file system, at which point it can find and run the initialization procedure, which is different for PC-BSD and Solaris.
- The initialization procedure runs other programs to find the file systems and start network and local service processes.

The term *booting* means bringing the operating system from a complete power-off condition to a steady-state, fully normal operating condition. It is worth noting here how a UNIX system is different from systemd in Linux, and launchd in OS X.

For both PC-BSD and Solaris family UNIX systems, the boot processes and graceful shutdown sequences are essentially the same, with just a minor difference in the program that handles system service initialization procedures.

23.2.5.2 Graceful Shutdown

For both PC-BSD and Solaris systems, graceful shutdown procedures are generally done as follows.

1. Shutdown system and user processes
2. Flush system memory to disk
3. Unmount file systems
4. Power off

These procedures may be subsumed under three distinct system call, system service phases in PC-BSD. The graceful shutdown or reboot procedures can be done graphically from the active window system, or from the command line or console window using the `halt` or `shutdown` commands and their options. An example of this on PC-BSD is as follows.

```
$ shutdown -h now
```

where **`-h`** means halt the processor, an **`now`** means immediately.

On Solaris, and example of using `shutdown` is a follows:

```
# shutdown -i0 -g180 "System down in 2 minutes"
```

where `-i0` means bring the system to the 0 run level, `-g180` means wait 2 minutes (180 seconds), and the message `System down in 2 minutes` is broadcast to all users.

23.2.5.3 Managing System Services

The two system service managers that are available in our base UNIX systems are init at boot time for PC-BSD, and SMF for Solaris. In this section, we cover introductory concepts and commands for both.

First, one question must be answered: Is a *service* a daemon?

The answer: Sometimes.

More accurately and generally, a service can be a process or collection of processes, the overall state of the system, or the state of a device, virtual device, dataset, etc. And as a provider of resources or a collection of an application's capabilities, it can have more than one instance. For example, many layered file systems, or multiple means of remote login to the system.

And that brings about an ancillary question as well: What is a daemon?

The answer: Basically a daemon is an ongoing background process that is not linked or controlled by a terminal, and particularly not connected in the usual way to standard output or stand error.

23.2.5.3.1 PC-BSD Overview of System Startup, Services, and Run Levels Most current LINUX distributions use *systemd* to start services, and OS X uses *launchd*. PC-BSD uses the traditional BSD-style *init*. Under the BSD-style init, there are no *run levels* (software configurations under which only a selected group of processes are running) and the file **/etc/inittab**, which controls the System V init protocol, does not exist. Instead, startup is controlled by rc scripts, as illustrated by the examples in Section 23.2.5.5.

But if init is run as a user process, it will emulate System V init behavior. It can invoke run levels, as System V init does. You can use init to change the run level of the system. A superuser can specify the desired run level on a command line, and init will signal the original (PID 1) init as displayed in Table 23.1.

To get a more detailed account of PC-BSD init, see the man page for init on your system.

When PC-BSD reaches the init stage of booting, **/etc/rc** reads **/etc/rc.conf** and **/etc/defaults/rc.conf** to determine which services are to be started. The specified services are then started by running the corresponding service initialization scripts located in **/etc/rc.d/** and **/usr/local/etc/rc.d/**. These scripts are similar to the scripts located in **/etc/init.d/** on LINUX systems.

The scripts found in **/etc/rc.d/** are for applications that are part of the *base* system, such as cron, sshd, and syslog. The scripts in **/usr/local/etc/rc.d/** are for user-installed applications such as Apache.

PC-BSD is a complete operating system, and user-installed applications are not considered to be part of the base system. User-installed applications are generally installed using packages or ports. In order to keep them separate from the base system, user-installed applications are installed under **/usr/local/**. User-installed binaries reside in **/usr/local/bin/**, configuration files are in **/usr/local/etc/**.

TABLE 23.1 Run Level Signal Emulation

Run-level	Signal	Action
0	`SIGUSR2`	Halt and turn the power off
1	`SIGTERM`	Go to single-user mode
6	`SIGINT`	Reboot the machine
c	`SIGTSTP`	Block further logins
q	`SIGHUP`	Rescan the **ttys(5)** file

Services are enabled by adding an entry for the service in **/etc/rc.conf**. The system defaults are found in **/etc/defaults/rc.conf** and these default settings are overridden by settings in **/etc/rc.conf**. Refer to **rc.conf** for more information about the available entries. When installing additional applications, review the application's install message to determine how to enable any associated services.

The following entries in a sample **/etc/rc.conf** enable sshd, enable Apache 2.4, and specify that Apache should be started with SSL.

```
# enable SSHD
sshd_enable="YES"
# enable Apache with SSL
apache24_enable="YES"
apache24_flags="-DSSL"
```

Once a service has been enabled in **/etc/rc.conf**, it can be started without rebooting the system:

```
# service sshd start
# service apache24 start
```

If a service has not been enabled, it can be started from the command line using `onestart`:

```
# service sshd onestart
```

23.2.5.3.2 Solaris System Startup, SMF, and Run Levels When Solaris boots and reaches the init phase, init on Solaris starts the svc.startd daemon, which is SMF (system management framework). SMF is the master process starter and restarter. Not only does SMF start services at boot time, but it also provides a mechanism to administer to services.

The Solaris init is the default first user process, or PID 1. Options given to the kernel during boot may result in an alternative first user process. The init process starts the core components of SMF and svc.configd. It restarts these components if they fail. For backward compatibility, init also starts and restarts general processes similar to PC-BSD init.

At any given time, the system is in one of eight possible run levels, similar to those defined previously for PC-BSD. Processes started by init for each of these run levels can be defined in **/etc/inittab**. The init process can be in one of eight run levels 0–6 and S or s. The run level changes when a privileged user runs **/usr/sbin/init**.

The run level S or s invoked by init switches the system into the single-user state. In this state, the system console device (**/dev/console**) is opened for reading and writing. Run levels 0, 5, and 6 are reserved states for shutting the system down. Run levels 2, 3, and 4 are available as multiuser operating states.

Solaris SMF manages applications and system services. It works in addition to the traditional UNIX startup scripts, init run levels, and configuration files. SMF works so that dependent services can automatically be restarted when necessary. Information needed to manage each service is stored in the service configuration repository, which provides a simplified way to manage each service. SMF also allows you to start or stop services that were not started by default at boot time. The examples shown for Solaris in Section 23.2.5.5 illustrate this.

SMF has a set of actions that can be invoked on a service by an administrator, generally the root user. These actions, which can be manually manipulated on the command line by, for example, the `svcadm` command, include enable, disable, refresh, and restart. Each service is managed by a service restarter, which carries out the administrative actions. In general, the restarters carry out actions by executing methods for a service. Methods for each service are defined in the service configuration repository. These methods allow the restarter to move the service from one state to another state.

The following is a listing of the commands used by SMF to manage services.

- `inetadm`: Controls the inetd daemon services
- `svcadm`: Does the common service management tasks, like enabling, disabling, or restarting service instances
- `svccfg`: Displays and makes changes to a service configuration repository
- `svcprop`: Provides shell script code by retrieving property values from a service configuration repository
- `svcs`: Gives verbose details of the service state of all instances in the service configuration repository
- `svcbundle`: Generates SMF manifests

See the following examples for a more complete description of using some of these commands. You can also examine the man pages on your system for any of these commands.

23.2.5.4 Examples of System Service Management

In this section we illustrate the use of both PC-BSD and Solaris service management by giving examples of how to enable server-side host services for the *telnet*, *ftp*, *rlogin* and *rsh* daemons.

In Chapter 25, we will show how you can implement these services securely inside of a PC-BSD Jail, a Solaris Zone, or within a VirtualBox VM.

In preparation for the PC-BSD components of this section, you should refer back to Chapter 20, Section 10, "The UNIX Super Server inetd" for more information on the inetd service operation and its configuration.

As will be noted, you should be cautious when executing these examples, particularly if you are not using one of the virtualization methodologies shown in Chapter 25, and are on an insecure network connected to the Internet. You must have root privileges on the computer to execute the following examples as well.

EXERCISE 23.3

What is the danger of using ftp, telnet, rlogin, or rsh from a remote site on the Internet on your home server?

23.2.5.4.1 Examples of Enabling telnet and ftp Servers on PC-BSD and Solaris

For the PC-BSD host or server:

- CAUTION: This step drops the firewall on the host computer for incoming traffic on ports 23 and 21.
- Use the PC-BSD Control Panel on the desktop and go to the Firewall Manager.
- In the Firewall Manager, add new TCP ports for telnet on 23 and ftp on 21, so incoming traffic can enter.
- At this point, restart the Firewall Manager and exit from it.
- Check to see if telnet is active:

```
$ netstat -anf inet|grep LIST|grep 23
tcp4       0      0  *.23
*.*                    LISTEN
```

- If you don't get this output, proceed with the following:

As superuser:

```
$ vi /etc/inetd.conf
```

- Uncomment , or remove the # sign as the first character in the lines:

```
ftp     stream   tcp nowait root  /usr/libexec/ftpd    ftpd-1
telnet  stream   tcp nowait root  /usr/libexec/telnetd telnetd
```

- Save the file.

- Enable inetd in **/etc/rc.conf** so you can initially start it in this session, and it always starts in every subsequent session, to enable telnet and ftp:

```
$ echo "inetd_enable="YES"" >> /etc/rc.conf
$ /etc/rc.d/inetd start
Starting inetd
```

- With the next command, you are basically *telneting* to your own machine to check telnet! You can check ftp the same way.
- Exit as superuser.

```
$ telnet localhost
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Trying SRA secure login:
User (bob):
Password: xxx
[ SRA accepts you ]
…
$ logout
```

- You're back at the original shell prompt.
- From a remote machine, *telnet* or *ftp* into the PC-BSD host you just enabled telnet and ftp on.

For Solaris host or server:

- By default, there are no firewall changes that have to be made.
- See if the telnet daemon is enabled and running.

```
bob@solaris:~$ svcadm enable network/telnet:default
svcadm: svc:/network/telnet:default: is not complete, missing
general/complete (see svcs -xv svc:/network/telnet:default for
details)
```

- Check with svcs.

```
bob@solaris:~$ svcs -xv svc:/network/telnet:default
svc:/network/telnet:default (?)
State: -
Reason: Service is incomplete, defined only by profile /etc/
svc/profile/generic.xml. To install this service, identify and
install the package which provides the service's primary
manifest. Use "pkg search 'svc\:/network/telnet:default'" to
```

```
identify the package, then "pkg install <pkg>" to install the
indicated package.
Impact: This service is not running.
```

- From the preceding two commands, we see it is not enabled or running, and why this is so!
- Do what is suggested, find out what the **pkg** is, and install the **pkg** as superuser.

```
bob@solaris:~$ su
Password: xxx
root@solaris:~# pkg search 'svc:/network/telnet:default'
INDEX                    ACTION VALUE
PACKAGE
org.opensolaris.smf.fmri set    svc\:/network/telnet\:default
pkg:/service/network/telnet@0.5.11-0.175.2.0.0.42.2
root@solaris:~# pkg install pkg:/service/network/
                telnet@0.5.11-0.175.2.0.0.42.2
           Packages to install:  1
            Services to change:  1
       Create boot environment: No
Create backup boot environment: No
DOWNLOAD            PKGS         FILES    XFER (MB)   SPEED
Completed            1/1         10/10    0.0/0.0     92.3k/s
PHASE                            ITEMS
Installing new actions           32/32
Updating package state database  Done
Updating package cache           0/0
Updating image state             Done
Creating fast lookup database    Done
Updating package cache           1/1
```

- Now that **pkg** has been installed, enable it.

```
root@solaris:~# svcadm enable network/telnet:default
```

- Exit root.

```
root@solaris:~# exit
exit
```

- Test telnet by logging on locally.

```
bob@solaris:~$ telnet 0
Trying 0.0.0.0...
```

```
Connected to 0.
Escape character is '^]'.
login: bob
Password: xxx
Last login: Fri Sep 26 05:28:33 from 192.168.0.12
Corporation      SunOS 5.11      11.2    June 2014
```

- It works! Log out.

```
bob@solaris:~$ logout
Connection to 0 closed by foreign host.
bob@solaris:~$
```

Enabling ftp and running it:

- Basically, repeat what was done for enabling telnet.
- Check if ftp is enabled and running.

```
bob@solaris:~$ svcadm enable network/ftp:default
svcadm: svc:/network/ftp:default: is not complete, missing
general/complete (see svcs -xv svc:/network/ftp:default for
details)
bob@solaris:~$ svcs -xv svc:/network/ftp:default
svc:/network/ftp:default (?)
State:-
Reason: Service is incomplete, defined only by profile /etc/
svc/profile/generic.xml. To install this service, identify and
install the package which provides the service's primary
manifest. Use "pkg search 'svc\:/network/ftp\:default'" to
identify the package, then "pkg install <pkg>" to install the
indicated package.
Impact: This service is not running.
```

- From the preceding two commands, we see it is not enabled or running, and why this is so!
- Do what is suggested, find out what the **pkg** is, and install the **pkg** as superuser.

```
bob@solaris:~$ su
Password: xxx
root@solaris:~#  pkg search 'svc\:/network/ftp\:default'
INDEX               ACTION VALUE                PACKAGE
org.opensolaris.smf.fmri set    svc:/network/ftp:default pkg:/
service/network/ftp@1.3.4.0.3-0.175.2.0.0.42.1
root@solaris:~# pkg install pkg:/service/network/
                ftp@1.3.4.0.3-0.175.2.0.0.42.1
```

```
           Packages to install:  1
            Services to change:  2
       Create boot environment: No
Create backup boot environment: No
DOWNLOAD              PKGS         FILES    XFER (MB)   SPEED
Completed             1/1          113/113  0.8/0.8     308k/s
PHASE                             ITEMS
Installing new actions            178/178
Updating package state database   Done
Updating package cache            0/0
Updating image state              Done
Creating fast lookup database     Done
Updating package cache            1/1
```

- Now that **pkg** has been installed, enable it.

```
root@solaris:~# svcadm enable network/ftp:default
```

- Get out of superuser.

```
root@solaris:~# exit
exit
```

- Now test it on the local host machine.

```
bob@solaris:~$ ftp 0
Connected to 0.0.0.0.
220 ::ffff:127.0.0.1 FTP server ready
Name (0:bob): <Enter>
331 Password required for bob
Password: xxx
230 User bob logged in
Remote system type is UNIX.
Using binary mode to transfer files.
ftp>
```

- It works! Get out of it locally.

```
ftp> quit
221 Goodbye.
bob@solaris:~$
```

- From a remote machine, `telnet` or `ftp` into the Solaris host you just enabled telnet and ftp on.

Examples of Enabling rlogin and rsh on a PC-BSD and Solaris Server.

For the PC-BSD host or server:

- CAUTION: This step drops the firewall on the host computer for incoming traffic on port 513.
- Use the PC-BSD Control Panel on the desktop and go to the Firewall Manager.
- In the Firewall Manager, add a new TCP port for `rlogin` and `rsh` on port 513, so incoming traffic can enter.
- At this point, restart the Firewall Manager and exit from it.
- Check to see if rlogin is active:

```
$ rlogin 0
```

- If your connection is refused, proceed with the following as superuser:

```
# vi /etc/inetd.conf
```

- Uncomment, or remove the # sign as the first character in the line:

```
login stream tcp nowait root /usr/libexec/rlogind rlogind
```

- Save the file.
- If you haven't enabled inetd for telnet and ftp, do the following:
- Enable inetd in **/etc/rc.conf** so you can initially start it in this session, and it always starts in every subsequent session, to enable rlogin and rsh:

```
# echo "inetd_enable="YES"" >> /etc/rc.conf
```

- Start, or restart, inetd.

```
# /etc/rc.d/inetd start
Starting inetd
```

- With the next command, you are basically using rlogin on your own machine to check the rlogin daemon! You can check rsh the same way.
- Exit as superuser.

```
$ rlogin localhost
connect to address ::1: Connection refused
Trying 127.0.0.1...
Password for bob@pc-2823: xxx
```

```
…
[bob@pcbsd-2823] ~%
```

- You are in! Log out.

```
[bob@pcbsd-2823] ~% logout
$
```

- You're back at the original shell prompt.
- From a remote machine, `rlogin` or `rsh` into the PC-BSD host you just enabled rlogin and rsh on.

For a Solaris server:

- Put the proper hostnames, IP addresses, and user permissions in your home directory's **.rlogin** file and also in the /etc/hosts.equiv file. See the man pages for rlogin and rsh for directions.
- Check to see if rlogin is enabled and running with the following two commands:

```
bob@solaris:~$ svcadm enable svc:/network/login:rlogin
svcadm: svc:/network/login:rlogin: is not complete, missing
general/complete (see svcs -xv svc:/network/login:rlogin for
details)
bob@solaris:~$ svcs -xv svc:/network/login:rlogin
svc:/network/login:rlogin (?)
State: -
Reason: Service is incomplete, defined only by profile /etc/
svc/profile/generic.xml. To install this service, identify and
install the package which provides the service's primary
manifest. Use "pkg search 'svc\:/network/login\:rlogin'" to
identify the package, then "pkg install <pkg>" to install the
indicated package.
Impact: This service is not running.
```

- Based on this, follow the instructions given in the `svcs` command as superuser.

```
bob@solaris:~$ su
Password: xxx
```

- Do a **pkg** search.

```
root@solaris:~# pkg search 'svc\:/network/login\:rlogin'
INDEX                        ACTION VALUE              PACKAGE
```

```
org.opensolaris.smf.fmri set    svc:/network/login:rlogin
pkg:/service/network/legacy-remote-utilities
@0.5.11-0.175.2.0.0.42.2
```

- Get that **pkg**!

```
root@solaris:~# pkg install pkg:/service/network/legacy-
                remote-utilities@0.5.11-0.175.2.0.0.42.2
           Packages to install:  1
            Services to change:  1
       Create boot environment: No
Create backup boot environment: No
DOWNLOAD                PKGS         FILES    XFER (MB)   SPEED
Completed               1/1          36/36    0.1/0.1     132k/s
PHASE                               ITEMS
Installing new actions              63/63
Updating package state database  Done
Updating package cache              0/0
Updating image state                Done
Creating fast lookup database       Done
Updating package cache              1/1
```

- Install rlogin.

```
root@solaris:~# svcadm enable svc:/network/login:rlogin
```

- Get out of root.

```
root@solaris:~# exit
exit
```

- Test rlogin on the local loopback.

```
bob@solaris:~$ rlogin 0
Password: xxx
Last login: Fri Sep 26 13:23:30 on rad/0
Corporation   SunOS 5.11  11.2  June 2014
bob@solaris:~$
```

- It works! Log out.

```
bob@solaris:~$ logout
Connection to 0 closed.
bob@solaris:~$
```

- Now test both rlogin and rsh from a remote client machine to the server you just enabled and ran rlogin on.

23.3 USER ADMINISTRATION

The two most important objectives of user administration are service and security: providing a service to ensure that the user base has access to and can fully take advantage of the resources that a modern UNIX system can provide, and securing the files and processes that the user base needs to utilize those resources. Later in this chapter we go over some of the security methodologies that a system administrator can deploy to keep the system secure. In Chapter 25, we will also show security methods that revolve around virtualization technologies such as Jails, Zones, and VirtualBox VMs.

The traditional UNIX technique for providing service and security is through access privileges, for the individual user, the group, or all others on the system, to specific objects on the system, such as files. We showed how to set access privileges on files in Chapter 5, Sections 5.4 and 5.5. Designing and implementing user groups, and access privileges for user groups, are the most important parts of this technique.

Certainly, user account creation and configuration is the first step in providing maximum service and security to the user base.

In the following section, we concentrate on how to manage user accounts and groups on both PC-BSD and Solaris. These activities are done a significant time after the initial installation of your UNIX system, but that does not mean that they cannot be done during initialization of the system. The examples in the following section illustrate:

- Two simple cases of text-based user account creation and configuration on PC-BSD
- A GUI account creation and configuration in PC-BSD
- A text-based account creation and configuration in Solaris

Adding a new user account to a UNIX system generally involves the following steps:

1. Assign the user a username, a user ID number, and a primary group, and decide which other groups the new user should be a member of. Enter this data into the system user account configuration files.
2. Assign a password to the new account.
3. Create a home directory for the user.
4. Place default or custom initialization files in the user's home directory.
5. Give the new user ownership of their home directory and initialization files.
6. Set other user account configuration details that would be useful for your system. This would include things like disk quotas and other resource limits, and system privileges.

7. Add the user to any other facilities in use as appropriate, such as the UNIX mail system and printing.
8. Grant or deny access to additional system resources as appropriate, using file protections or the resources' own internal mechanisms (e.g., the **/etc/ftpusers** file lists those users denied access to the ftp service).
9. Perform any other site-specific initialization tasks.
10. Test the new account.

EXERCISE 23.4

Given these 10 steps, make a table or chart of what users and groups need to be added to your system, and what their default account parameters and group memberships should be. What command can you use to identify all existing groups on the system?

23.3.1 Adding and Deleting a User in a Text-Based Interface on PC-BSD

The following are two simple examples of how to add a new user to PC-BSD in a text-based command line interface. The first one is an interactive session that uses the `adduser` command to add a single user account. The second one uses `adduser` in *batch* mode, in conjunction with a file that contains a listing of several user accounts that can be added all at one time. An abbreviated listing of the syntax of the `adduser` command is as follows.

SYNTAX

`adduser [options] [option argument(s)]`

Purpose: The **`adduser`** utility is a shell script for adding new users. It creates `passwd/group` entries, a home directory, copies dotfiles and sends the new user a welcome message. It supports two modes of operation. It may be used interactively at the command line to add one user at a time, or it may be directed to get the list of new users from a file and operate in batch mode without requiring any user interaction.

Commonly used options/features:

`-C` Interactively create the **/etc/adduser.conf** file, whose contents become a template for new user creation. By default, this file does not exist. But if you execute **`adduser -C`** , you will be able to interactively design the contents of this file.

`-f filename` Enter batch mode, where the contents of **`filename`** are used to create multiple user accounts. When the **`-f`** option is used, the user account information must be stored in a specific format. All empty lines or lines beginning with a # will be ignored. All other lines must contain ten fields separated by colons (:), as will be described. Command line options do not take precedence over values in the fields. Only the password field may contain a colon as part of the string.

`name:uid:gid:class:change:expire:gecos:home _ dir:shell:password`

`name` Login name. This field may not be empty.

`uid` Numeric login user ID. If this field is left empty, it will be automatically generated.

gid Numeric primary group ID. If this field is left empty, a group with the same name as the user name will be created and its GID will be used instead.
class Login class. This field may be left empty.
change Password ageing. This field denotes the password change date for the account. The format of this field is the same as the format of the **-p** argument to **pw**. It may be **dd-mmm-yy[yy]**, where **dd** is for the day, **mmm** is for the month in numeric or alphabetical format (e.g., **10** or **Oct**), and **yy[yy]** is the four- or two-digit year. To denote a time relative to the current date the format is: **+n[mhdwoy]**, where **n** denotes a number, followed by the minutes, hours, days, weeks, months or years after which the password must be changed. This field may be left empty to turn it off.
expire Account expiration. This field denotes the expiry date of the account. The account may not be used after the specified date. The format of this field is the same as that for password ageing. This field may be left empty to turn it off.
gecos Full name and other extra information about the user.
home _ dir Home directory. If this field is left empty, it will be automatically created by appending the username to the home partition. The **/nonexistent** home directory is considered special and is understood to mean that no home directory is to be created for the user.
shell Login shell. This field should contain either the base name or the full path to a valid login shell.
password User password. This field should contain a plaintext string, which will be encrypted before being placed in the user database. If the password type is `yes` and this field is empty, it is assumed the account will have an empty password. If the password type is `random` and this field is not empty, its contents will be used as a password. This field will be ignored if the **-w** option is used with a `no` or `none` argument. Be careful not to terminate this field with a closing colon (**:**) because it will be treated as part of the password.

The following example illustrates how to interactively create a single user account from the command line as superuser. If you make a mistake in creating a user account, you can always remove the account immediately by using the `rmuser` command.

Example 23.1: PC-BSD adduser Command for a Single User Account

```
[bob@pcbsd-1727] ~% su
password: xxx
[bob@pcbsd-1727] /usr/home/bob# adduser
Username: sarwar
Full name: Mansoor Sarwar
Uid (Leave empty for default):
Login group [bob]:
Login group is bob. Invite sarwar into other groups? []:
Login class [default]:
Shell (sh csh tcsh git-shell bash rbash nologin) [sh]:
Home directory [/home/sarwar]:
```

```
Home directory permissions (Leave empty for default):
Use password-based authentication? [yes]:
Use an empty password? (yes/no) [no]:
Use a random password? (yes/no) [no]:
Enter password: yyy
Enter password again: yyy
Lock out the account after creation? [no]:
Username   : sarwar
Password   : ***
Full Name  : Mansoor Sarwar
Uid        : 1002
Class      :
Groups     : bob
Home       : /home/sarwar
Home Mode  :
Shell      : /bin/sh
Locked     : no
OK? (yes/no): yes
adduser: INFO: Successfully added (sarwar) to the user database.
Add another user? (yes/no): no
Goodbye!
[bob@pcbsd-1727] /usr/home/bob#
```

EXERCISE 23.5

Create a single new account on your PC-BSD UNIX system, with the `adduser` command. Be sure to use the entries you made in your answer to Exercise 23.4 to override the defaults for user account configurations on your system.

The following example shows how to create multiple user accounts simultaneously by using the `adduser -f filename` command. It assumes you have a group named **bob** on your system, and that you have superuser privilege. If you don't have a group named **bob** on your system, use any other existing group that fits the service and security requirements of your system for this example. If you make a mistake in creating user accounts, you can always remove the accounts immediately by using the `rmuser` command, either individually or in batch mode.

Example 23.2: PC-BSD `adduser` Batch Mode User Account Creation

1. Use your favorite text editor to create a file named **addusertest** in your home directory.

   Then, enter the following single line of text, exactly as shown, into the file **addusertest**:

   `tob::bob:::AMR:/usr/home/tob:/bin/csh:xxx`

   Save the file and quit the text editor.

According to the `adduser -f` option requirements, as shown previously and in the `adduser` man pages on your system, you need to supply 10 fields separated by colons (:) on a line. For the line you just entered into `addusert-est`, the meanings of each entry into these 10 fields are as follows (`blank` means nothing between the colon characters).

`tob`	The new username.
blank	The UID, when left blank, is automatically assigned.
`bob`	The primary GID, in our case **bob**.
blank	The login class, in our case blank.
blank	Password aging, in our case blank, meaning turned off.
blank	Account expiration, in our case blank, meaning turned off.
`AMR`	The account *gecos*, or full name and description.
`/usr/home/tob`	The user account home directory, **/usr/home/tob**.
`/bin/csh`	The default login shell, in our case the C shell.
`xxx`	A password, literally `xxx` in our case.

2. As superuser, type the following on the command line:
```
# adduser -f addusertest
```
Your system will respond that you have successfully added a user account named **tob** to the user database.
3. To test this new user account, log off your system and log on as **tob** (**AMR**) with the password `xxx`.

EXERCISE 23.6

According to the table or chart of user accounts and their configuration requirements you did for Exercise 23.4, create a file of new user accounts. As shown in Example 23.2, for each user account you must have a single line in the file with 10 colon-delimited fields that contain the configuration for each user. The table or chart design determines what is in the file.

23.3.1.1 Using `rmuser` to Delete Users

To completely remove a user from the system, use the command `rmuser` as the superuser. This command performs the following steps:

a. Removes the user's crontab entry, if one exists.

b. Removes any `at` jobs belonging to the user.

c. Kills all processes owned by the user.

d. Removes the user from the system's local password file.

e. Optionally removes the user's home directory, if it is owned by the user.

f. Removes the incoming mail files belonging to the user from **/var/mail**.

g. Removes all files owned by the user from temporary file storage areas such as **/tmp**.

h. Finally, removes the username from all groups to which it belongs in **/etc/group**. If a group becomes empty and the group name is the same as the username, the group is removed. This complements the per-user unique groups created by `adduser`.

`rmuser` cannot be used to remove superuser accounts.

By default, an interactive dialog mode is used, as shown in the following example.

```
# rmuser amy
Matching password entry
amy:*:1001:1001::0:0:J. Random User:/home/amy:/usr/local/bin/zsh
Is this the entry you wish to remove? yes
Remove user's home directory (/home/amy)? yes
Removing user (amy): mailspool home passwd.
#
```

For more information on the `rmuser` command, see the man page for `rmuser` on your system.

23.3.1.2 Creating, Modifying, and Deleting Groups Using a Text-Based Interface in PC-BSD

We also provide an abbreviated syntax description of the pw command, showing its group subcommand options. This command allows you to use a text-based interface to manage (create, remove, modify, and display) groups on the system.

SYNTAX

```
pw [-R rootdir] [-V etcdir] groupadd [group|gid] [-C config] [-q] [-n
group] [-g gid] [-M members] [-o] [-h fd | -H fd] [-N] [-P] [-Y]
pw [-R rootdir] [-V etcdir] groupdel [group|gid] [-n name] [-g gid] [-Y]
pw [-R rootdir] [-V etcdir] groupmod [group|gid] [-C config] [-q] [-n
name] [-g gid] [-l name] [-M members] [-m newmembers] [-d oldmembers]
[-h fd | -H fd] [-N] [-P] [-Y]
pw [-R rootdir] [-V etcdir] groupshow [group|gid] [-n name] [-g gid]
[-F] [-P] [-a]
pw [-R rootdir] [-V etcdir] groupnext [-C config] [-q]
pw [-R rootdir] [-V etcdir] lock [name|uid] [-C config] [-q]
pw [-R rootdir] [-V etcdir] unlock [name|uid] [-C config] [-q]
```

Purpose:

The **pw** utility is a command-line-based editor for the system user and group files, allowing the superuser an easy-to-use and standardized way of adding, modifying, and removing users and groups. Note that **pw** only operates on the local user and group files. NIS users and groups must be maintained on the NIS server. The **pw** utility handles updating the passwd, master.passwd, group, and the secure and insecure password database files, and must be run as root. The first keyword option to **pw** on the command line provides the context for the remainder of the arguments. The keywords **user** and **group** may be combined with **add**, **del**, **mod**,

show, or **next** in any order. (For example, **showuser**, **usershow**, **show user**, and **user show** all mean the same thing.) This flexibility is useful for interactive scripts calling **pw** for user and group database manipulation. Following these keywords, you may optionally specify the group name or numeric ID as an alternative to using the **-n name**, **-u uid**, or **-g gid** options.

Output: Modified user or group profiles and information on the system

Options and Option Arguments:

The following options are common to most or all modes of operation:

-R rootdir Specifies an alternate root directory within which **pw** will operate. Any paths specified will be relative to **rootdir**.

-V etcdir This flag sets an alternate location for the password, group and configuration files, and may be used to maintain a user/group database in an alternate location. If this switch is specified, the system **/etc/pw.conf** will not be sourced for default configuration data, but the file **pw.conf** in the specified directory will be used instead (or none, if it does not exist). The **-C** flag may be used to override this behavior. As an exception to the general rule where options must follow the operation type, the **-V** flag must be used on the command line before the operation keyword.

-C config By default, **pw** reads the file **/etc/pw.conf** to obtain policy information on how new user accounts and groups are to be created. The **-C** option specifies a different configuration file. While most of the contents of the configuration file may be overridden via command line options, it may be more convenient to keep standard information in a configuration file.

-q Use of this option causes **pw** to suppress error messages, which may be useful in interactive environments where it is preferable to interpret status codes returned by **pw** rather than messing up a carefully formatted display.

-N This option is available in add and modify operations, and tells **pw** to output the result of the operation without updating the user or group databases. You may use the **-P** option to switch between standard passwd and readable formats.

-Y Using this option with any of the update modes causes **pw** to run **make(1)** after changing to the directory **/var/yp**. This is intended to allow automatic updating of NIS database files. If separate passwd and group files are being used by NIS, then use the **-y** path option to specify the location of the NIS passwd database so that **pw** will concurrently update it with the system password databases.

Group Options:

The **-C** and **-q** options (explained at the start of the previous section) are available with the group manipulation commands. Other common options to all group-related commands are:

-n name Specify the group name.

-g gid Specify the group numeric ID. As with the account name and ID fields, you will usually only need to supply one of these, as the group name implies the UID and vice versa. You will only need to use both when setting a specific group ID against a new group or when changing the UID of an existing group.

-M memberlist This option provides an alternative way to add existing users to a new group (in **groupadd**) or replace an existing membership list (in **groupmod**). **memberlist** is a comma-separated list of valid and existing user names or UIDs.

-m newmembers Similar to **-M**, this option allows the addition of existing users to a group without replacing the existing list of members. Login names or user IDs may be used, and duplicate users are silently eliminated.

-d oldmembers Similar to **-M**, this option allows the deletion of existing users from a group without replacing the existing list of members. Login names or user IDs may be used and duplicate users are silently eliminated.

groupadd also has a **-o** option that allows allocation of an existing group ID to a new group. The default action is to reject an attempt to add a group, and this option overrides the check for duplicate group IDs. There is rarely any need to duplicate a group ID.

The **groupmod** command adds one additional option:

-l name This option allows changing of an existing group name to **name**. The new name must not already exist, and any attempt to duplicate an existing group name will be rejected.

Options for **groupshow** are the same as for **usershow**, with **-g gid** replacing **-u uid** to specify the group ID. The **-7** option does not apply to the **groupshow** command.

The command **groupnext** returns the next available group ID on standard output.

For a summary of options available with each command, you can use:

pw [command] help

For example, **pw useradd help** lists all available options for the **useradd** operation.

Command Arguments: None

Examples

1. Adding a group:

```
# pw groupadd project0
# pw groupshow project0
project0:*:1100:
```

   In this example, 1100 is the GID of **project0**. Right now, **project0** has no members.

2. Adding user accounts to a new group:

```
# pw groupmod project0 -M mansoor
# pw groupshow project0
project0:*:1100:mansoor
```

   These commands will add **mansoor** as a member of **project0** and show his UID number.

   The argument to -M is a comma-delimited list of users to be added to a new (empty) group or to replace the members of an existing group. To the user, this group membership is different from (and in addition to) the user's primary group listed in the password file. This means that the user will not show up as a member when using the groupshow subcommand, but will show up when the information is queried via the id command or a similar tool. When pw is used to add a user to a group, it only manipulates **/etc/group** and does not attempt to read additional data from **/etc/passwd**.

3. Adding a new member to a group:

```
# pw groupmod project0 -m bob
# pw groupshow project0
project0:*:1100:mansoor,bob
```

4. Using the id command to determine group membership:

```
% id mansoor
uid=1001(mansoor) gid=1001(mansoor) groups=1001(mansoor),
1100(project0)
```

In this example, **mansoor** is a member of the groups **mansoor** and **project0**.

23.3.2 Adding/Deleting and Maintaining Users and Groups in a GUI-Based Interface on PC-BSD

The easiest and most efficient way to create and manage user accounts and groups on PC-BSD is by using the GUI-based User Manager. The first step in launching the User Manager is to open a root window from the Kickoff Application Launcher in the lower-left corner of the screen, under **Applications>Utilities>Root Terminal**. Then, in the terminal window that appears, type the following, as shown in Figure 23.1:

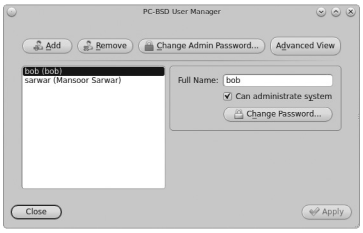
```
# pc-usermanager
```

The system then launches the User Manager in its own window, as shown in Figure 23.2.

If you click the Remove button for a highlighted user, a pop-up menu will ask if you would like to also delete the user's home directory (along with all of their files). If you click No, the user will still be deleted but their home directory will remain. If you have only created one user account, the Remove button will be grayed out as you need at least one user to be able to login to the PC-BSD system.

```
root@pcbsd-1727:~
[root@pcbsd-1727] ~# pc-usermanager
```

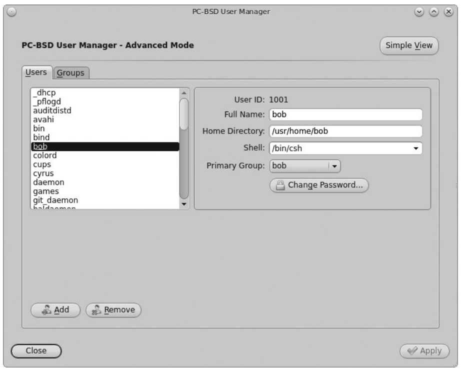
FIGURE 23.1 Root terminal window.



bob (bob)
sarwar (Mansoor Sarwar)

Full Name: bob
☑ Can administrate system

FIGURE 23.2 User Manager window.



_dhcp
_pflogd
auditdistd
avahi
bin
bind
bob
colord
cups
cyrus
daemon
games
git_daemon

User ID: 1001
Full Name: bob
Home Directory: /usr/home/bob
Shell: /bin/csh
Primary Group: bob

FIGURE 23.3 `User Manager – Advanced Mode` window.

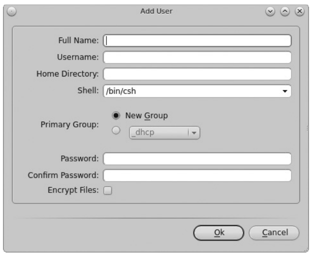
While a removed user will no longer be listed, the user account will not actually be deleted until you click the `Apply` button. A pop-up message will indicate that you have pending changes if you close User Manager without clicking `Apply`. If you change your mind, click `No` and the user account will not be deleted; otherwise, click `Yes` and the user will be deleted and User Manager will close.

The password for any user can be changed by first highlighting the user name then clicking the `Change Password` button. You will not be prompted for the old password in order to reset a user's password; this can be handy if a user has forgotten their password and can no longer log into the PC-BSD system. If you click the `Change Admin Password` button, you can change the root user's password.

If you click the `Advanced View` button, this screen will change to show all of the accounts on the system, not just the user accounts that you created. An example is seen in Figure 23.3.

The accounts that you did not create are known as system accounts and are needed by the operating system or installed applications. You should not delete any accounts that you did not create yourself, as doing so may cause a previously working application to stop working. Advanced View provides additional information associated with each account, such as the user ID number, full name (description), home directory, default shell, and primary group.

Figure 23.4 shows the `Add User Account Creation` screen that opens when you click the `Add` button under `Advanced View`.



Full Name:
Username:
Home Directory:
Shell: /bin/csh
Primary Group: [✓ New Group] _dhcp
Password:
Confirm Password:
Encrypt Files: ☐

FIGURE 23.4 `Add User` window.

This screen is used to input the following information when adding a new user or system account:

- Full Name: This field provides a description of the account and can contain spaces. If it is a user account, use the person's first and last name. If it is a system account, input a description to remind you which application uses the account.

- Username: The name the user will use when they log in to the system; it is case sensitive and can not contain any spaces. If you are creating a system account needed by an application, use the name provided by the application's installation instructions. If the name that you choose already exists as an account, it will be highlighted in red and the utility will prompt you to use another name.

- Home Directory: Leave this field empty for a user account as the system will automatically create a ZFS dataset for the user's home directory under **/usr/home/username**. We supply more information on the structure and administration of the ZFS file system in Chapter 24. However, if you are creating a system account it is important to override this default by typing in **/var/empty** or **/nonexistent** unless the application's installation instructions specify that the account needs a specific home directory.

- Shell: This drop-down menu contains the shells that are available to users when they are at a command prompt. You can either keep the default or select a shell which the user prefers.

- Primary Group: If you leave the default button of New Group selected, a group will be created with the same name as the user. This is usually what you want unless you are creating a system account and the installation instructions specify a different group name. The drop-down menu for specifying a group name will only show existing groups, but you can quickly create a group using the Groups tab.

- Password: The password is case sensitive and needs to be confirmed.

- Encrypt Files: If this box is selected, the user's home directory will automatically be encrypted with PEFS. When the user logs in, the contents of their home directory are automatically decrypted after they enter their password. When they logout, the contents of their home directory are automatically encrypted and will appear as gibberish to other users who do not know the password. For this reason, it is important

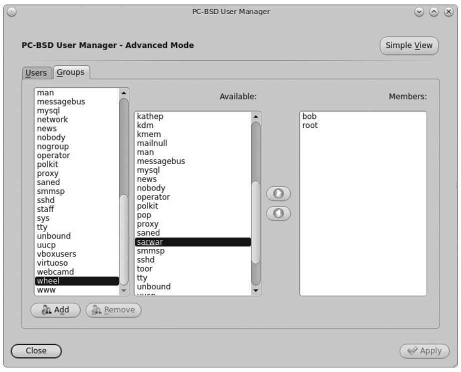


FIGURE 23.5 User Manager `Groups` tab window.

to select a good password that the user will not forget. At this time, there is no easy mechanism for changing the user's password if their home directory is encrypted.

Once you have made your selections, press the OK button to create the account.

In the Advanced View of the User Manager, if you click the `Groups` tab, you are presented with various ways of managing groups, as seen in Figure 23.5

This screen has three columns:

Column 1: Groups: shows all of the groups on the system.

Column 2: Available: shows all of the system and user accounts on the system in alphabetical order.

Column 3: Members: indicates if the highlighted group contains any user accounts.

To add an account to a group, highlight the group name in the first column. Then, highlight the account name in the **Available** column. Click the right arrow and the selected account will appear in the **Members** column. You should only add user accounts to groups that you create yourself or when an application's installation instructions indicate that an account needs to be added to a group.

If you click the `Add` button, a pop-up menu will prompt you for the name of the new group. Once you press OK, the group will be added to the **Groups** column.

If you click the `Remove` button, the highlighted group will automatically be deleted after you press the `Apply` button, so be sure to do this with care. Again, do not remove any groups that you did not create yourself or applications that used to work may stop working.

EXERCISE 23.7

Scenario 1: You have an air-gapped PC-BSD computer in a room. You want only certain users in a defined group named **proj** to be able to access files in a directory named **proj1** in your account on the file system of that computer. Use the graphical User Manager to add those certain users, and yourself, to the group **proj**. Create the **proj1** directory. Then, set the permission bits appropriately on your home directory, the **proj1** directory, and any files put in that directory, so that only those certain users can read, write, and execute the files in it. The room is open to the public, but to login to the computer, each individual user has to use their own password.

Scenario 2: There is another air-gapped computer in the room, older, but it still runs the latest PC-BSD! Unfortunately, it does not have a graphics-capable monitor. It can only use a text-based command line interface. Achieve the same results as Scenario 1, but by using only text-based commands. Name the group **project2**, and the directory **proj2**.

23.3.3 Adding/Deleting and Managing Users and Groups in a Text-Based Interface on Solaris

The following examples will create and manage user accounts using the command line on Solaris. However, the User Manager panel available through the **pkg:/system/management/visual-panels/panel-usermgr** IPS package can also be used. This GUI is much easier to use and can be started using one of the following methods:

From the desktop, choose **System>Administration>User Manager**.

Start the User Manager GUI from the command line by typing **`vp usermgr &`**

Solaris supplies the user administration commands described in Table 23.2 for setting up and managing user accounts from the command line.

23.3.3.1 Adding User Accounts

You can add new user accounts on the local system by using the `useradd` command. This command adds an entry for the new user into the **/etc/passwd** and **/etc/shadow** files.

The syntax for the `useradd` command is as follows, and Table 23.3 describes the options in more detail.

SYNTAX

```
useradd [option(s)] [option argument(s)] <loginname>
```

Purpose: To add a user account to the system

Commonly used options/features:

`-D` See default values for user account configuration

`-k` Copy initialization files from **/etc/skel** into user account

TABLE 23.2 Account Administration Commands

Command	Description
`useradd`	Add a new user account
`userdel`	Delete a user account
`usermod`	Modify a user account
`groupadd`	Add a new group
`groupmod`	Modify a group

TABLE 23.3 `useradd` Command Options

Option	Description
`-A <authorization>`	One or more comma-separated authorizations.
`-b <base-dir>`	Specifies the default base directory for the system, unless `-d` is not specified.
`-u <uid>`	Sets the unique UID for the user.
`-o`	Allows the UID to be duplicated. The default is to not allow a duplicate UID.
`-g <gid>`	Specifies a predefined GID as the user's primary group.
`-G <gid>`	Specifies the new user's secondary group.
`-m`	Creates a new home directory if one does not already exist.
`-s <shell>`	Defines the full pathname to the user's login shell. The default is **/bin/sh**.
`-c <comment>`	Specifies user's full name, location, phone number in a comment.
`-d <dir>`	Specifies the home directory of the new user.
`-D`	Displays the default values for **group**, **basedir**, **skel-dir**, etc.

Many additional options are available, although most of them are not used as often as the ones in Table 23.3. You can refer to the man pages to find a listing of all the options to the `useradd` command.

The `useradd` command can be used with just one argument and no options, as follows:

```
# useradd sarwar
```

This creates the user account named **sarwar** using all default options. To see all of the default values, type:

```
# useradd -D
group=staff,10 project=default,3 basedir=/export/home
skel=/etc/skel shell=/usr/bin/bash inactive=0
expire= auths= profiles= roles= limitpriv=
defaultpriv= lock_after_retries=
#
```

These defaults can be modified by using the `-D` option with the `useradd` command as follows:

```
# useradd -D -b /apache24/home
```

The defaults for the `useradd` command are stored in the **/usr/sadm/defadduser** file. This file does not initially exist and is created the first time the command is executed using the `-D` option. This file can then be edited manually or modified. For example, to change the default group, execute the following command:

```
# useradd -D -g development
```

You can also temporarily override the default options by specifying them on the command line when executing the `useradd` command. The following example creates a new login account for **sarwar**:

```
# useradd -u 4000 -g other -d /export/home/sarwar -m -s /usr/bin/
bash -c "Mansor Sarwar, ext. xxx" sarwar
```

The login name is **sarwar**, the UID is **3000**, and the group is **other**. In this example, you instruct the system to create a home directory named **/export/home/sarwar**. The default shell is **/usr/bin/bash**, and the initialization files are to be copied from the **/etc/skel** directory.

Use the `-k` option to the `useradd` command to copy all of the user initialization files found in the **/etc/skel** directory into the new user's home directory. This directory can be customized or changed by specifying an alternate directory containing the files to be copied by supplying an argument to the `-k` option.

23.3.3.2 Assigning a UID

If the `-u` option is not used to specify a UID, the UID defaults to the next available number above the highest number currently assigned. For example, if UIDs 100, 110, and 200 are already assigned to login names, the next UID that is automatically assigned is 201.

23.3.3.3 Password Management and Creation

After creating the new account with the `useradd` command, the user account has not yet been activated and cannot be used. The next step is to set a password for the account using the passwd command as follows:

```
# passwd sarwar
passwd: Changing password for sarwar
New Password: yyy
Re-enter new Password: yyy
passwd: password successfully changed for sarwar
#
```

Some of the more common options that can be used with the `passwd` command are described in Table 23.4.

For a complete listing of the options, refer to the online man pages for the `passwd` command.

To force a user to change his or her password at the next login, type:

```
# passwd -f sarwar
passwd: password information changed for sarwar
#
```

TABLE 23.4 `passwd` Options

Option	Description
`-s <name>`	Shows password attributes. Along with `–a`, shows all users accounts.
`-d <name>`	Deletes password for name and unlocks the account.
`-e <name>`	Changes the login shell, in **/etc/passwd** file, for a user.
`-f <name>`	Expires the password; forces change of password at next login.
`-h <name>`	Changes the home directory, in **/etc/passwd** file, for user.
`-l <name>`	Locks a user's account. Use `-d` or `-u` to unlock.
`-N <name>`	Makes the password entry for `<name>` a value that cannot be used for login.
`-u <name>`	Unlocks an account.

To change a user's home directory, type

```
# passwd -h sarwar
```

The system responds with the following:

```
Default values are printed inside of '[ ]'.
To accept the default, type <return>.
To have a blank entry, type the word 'none'.
Home Directory [/home/sarwar]: /home/smsarwar
passwd: password information changed for smsarwar
```

23.3.3.4 Modifying User Accounts

You can use the `usermod` command to modify existing user accounts, and you can also change most of the options that were used when the account was originally created.

The following example changes the login name for user **sarwar** to **smsarwar**:

```
# usermod -d /home/smsarwar -m -s /bin/zsh -l smsarwar sarwar
```

This example also changed the home directory to **/home/smsarwar** and the default shell to **/bin/zsh**.

23.3.3.5 Modifying the Home Directory

When you're changing the home directory, unless the `-d` and `-m` options are used, existing files still must be manually moved from the old home directory to the new home directory. In all cases, symbolic links, application-specific configuration files, and various other references to the old home directory must be manually updated.

To set a user's account expiration date, enter the following:

```
# usermod -e 11/15/2022 smsarwar
```

The account is now set to expire October 15, 2022. Notice the entry made to the **/etc/shadow** file:

```
smsarwar:$5$VWZPYaFo$EVDwt4h/NVxegGOjNI0jA99G/U.6aQTlVdAOc.
GGin3:15593:::::15993:
```

The last field for the **smsarwar** entry in the **/etc/shadow** file is 15993. This is an absolute date expressed as the number of days since the UNIX epoch (January 1, 1970). When this number is reached, the login is expired.

23.3.3.6 Deleting User Accounts

You can use the `userdel` command, similar to PC-BSD `rmuser`, to delete a user's login account from the system. You can specify options to save or remove the user's home directory. The syntax for the `userdel` command is as follows.

```
userdel [-r] [-S repository] <login-name>
```

The option `-r` removes the user's home directory from the local file system. If this option is not specified, only the login is removed; the home directory remains intact.

Make sure you know where the user's home directory is located before removing it. Some users have / as their home directory, and removing their home directory would remove important system files.

The following example removes the login account for **smsarwar** but does not remove the home directory:

```
# userdel smsarwar
```

23.3.3.7 Adding Group Accounts

You use the `groupadd` command to add new group accounts on the local system. This command adds an entry to the **/etc/group** file. The syntax for the `groupadd` command is as follows.

```
groupadd [-g gid [-o]] [-S repository] [-U user1[,user2..] ] group
```

Table 23.5 describes the `groupadd` command options.

The following example adds to the system a new group named acct with a GID of 1000:

```
# groupadd -g 1000 acct
```

23.3.3.8 Assigning a GID

If the `-g` option is not used to specify a GID, the GID defaults to the next available number above the highest number currently assigned. For example, if GIDs 100, 110, and 200 are already assigned to group names, the next GID that is automatically assigned is 201.

TABLE 23.5 `groupadd` Command Options

Option	Description
`-g <gid>`	Assigns the GID `<gid>` for the new group
`-o`	Allows the GID to be duplicated
`-S`	Specifies which name service repository will be updated
`-u`	Adds a list of users to a group

23.3.3.9 Modifying Group Accounts

You use the `groupmod` command to modify the definitions of a specified group. The syntax for the `groupmod` command is as follows.

```
groupmod [-S [files | ldap]] [-U user1[,user2]] [-g gid [-o]] [-n
name] group
```

Table 23.6 describes the `groupmod` command options.

The following example changes the **engrg** group GID from 200 to 2000:

```
# groupmod -g 2000 engrg
```

Any files that had the group ownership of **engrg** are now without a group name. A long listing would show a group ownership of 200 on these files, the previous GID for the **engrg** group. The group 200 no longer exists on the system, so only the GID is displayed in a long listing.

23.3.3.10 Deleting Group Accounts

You use the `groupdel` command to delete a group account from the local system. The syntax for the `groupdel` command is as follows.

```
groupdel [-S repository] <group-name>
```

The `-S` option allows the name service to be specified. The valid repositories that can be specified are **files** and **ldap**. When the repository is not specified, `groupdel` reads the file **nsswitch.conf**.

TABLE 23.6 `groupmod` Command Options

Option	Description
`-g <gid>`	Assigns the new GID `<gid>` for the group.
`-o`	Allows the GID to be duplicated.
`-n <name>`	Specifies a new name for the group.
`-S`	Valid repositories are files and **ldap**. The repository specifies the name service to be updated
`-U`	Updates the list of users for the group as follows: + adds, - removes, null replaces.

The following example deletes the group named **acct** from the local system:

```
# groupdel acct
```

EXERCISE 23.8

You have an air-gapped Solaris computer in a room. You want only certain users in a defined group named **proj** to be able to access files in a directory named **proj1** in your account on the file system of that computer. Use the text-based commands shown previously to add those certain users, and yourself, to the group **proj**. Create the **proj1** directory. Then, set the permission bits appropriately on your home directory, the **proj1** directory, and any files put in that directory, so that only those certain users can read, write and execute the files in it. The room is open to the public, but to login to the computer, each individual user has to use her own password.

23.4 ADDING A HARD DISK TO THE SYSTEM

A common task done by a UNIX system administrator is to add various hardware components any significant time after the system has been installed and configured. To help the administrator of a UNIX system with that task, the following sections will detail:

- Some preliminary considerations that need to be made, such as the availability of software device drivers for the new hardware to be added
- How the hardware will be recognized, configured and deployed on the system
- Identification of possible paths to replacement and upgrading of existing hardware

In this section, we will illustrate how to add disk drives for use with the ZFS file system, and for other file system types as well. ZFS is described completely in Chapter 24. The additions will be done for SATA hard disks installed internally, and USB removable media such as external, enclosure-mounted hard drives.

Generally, when you add a SATA hard disk inside the computer case some significant time after you have installed the operating system on the computer, you will want to format it to ZFS. But when you add an external USB medium, such as a thumb drive or hard disk, it will already be formatted to FAT32 (in the case of most popular commercially available USB thumb drives), or to some other format depending on the hard disk media. You will format either kind of media to ZFS, as shown in Chapter 24.

If all you want to use a FAT32-formatted USB thumb drive for is to transfer files (such as text files, C program code, LibreOffice documents, etc.), to and from the computer, then the automatic mounting (in Solaris) and semiautomatic mounting (in PC-BSD) of USB thumb drives will help you to accomplish this. In both systems, unmounting of USB thumb drives is done manually. In Solaris, right-click on the USB thumb drive icon on the Gnome desktop, and make the choice **Unmount**. In PC-BSD, use the Mount Tray icon, and make the **Unmount** choice for the USB thumb drive.

When a USB thumb drive is automatically mounted on Solaris or PC-BSD, the path to it is **/media**.

It is possible, though, to add a removable medium for use with ZFS to either PC-BSD or Solaris, and we actually do this in Chapter 24, Section 24.2.2, Examples 24.5 and 24.6. But you must be cautious when creating zpools and ZFS datasets on removable media and then removing the media without properly unmounting the ZFS datasets. These procedures are addressed more fully in Chapter 24.

23.4.1 Preliminary Considerations when Adding New Disk Drives

If you insert a USB thumb drive that you know is functioning properly into your computer running either Solaris or PC-BSD, and it doesn't automount or is not recognized, the chances are that your UNIX system does not have a device driver available to enable communication between the computer and the thumb drive.

The same is true when you connect a SATA bus hard drive properly, but the probability of this is much lower. The best and easiest thing to do in a case like this is to use another USB or SATA device. Both Solaris and PC-BSD have facilities to find and install device drivers on your system for a device, but this process is time consuming and may not be fruitful for the particular device in question. Also, it is possible to write a driver for your device, which is even more time consuming. The important thing here is not that the USB thumb drive is formatted to FAT32, but that a manufacturer has the device drivers available automatically when their device is inserted. This is not always true.

It is important to know the physical device name, the instance name, and the logical device name of disk drives on your system, but practically speaking, for the system administrator, easily finding out the logical device name of a disk drive is most important. This is because in the ZFS file system on Solaris and PC-BSD, the logical device name is what you use to create zpools, the critical first step in using ZFS. This is shown in full detail in Chapter 24.

You may want to add a hard disk to your UNIX system that has been used on another computer operating system previously. In that case, the following examples can be deployed to repartition and prepare that hard disk for new use on your UNIX system. Because ZFS is a volume manager, it is also possible to deploy a disk drive formatted and sliced for another operating system, and use ZFS on it simultaneously with the other operating system partitions.

23.4.2 A Quick and Easy Way to Find Out the Logical Device Names of Disks Actually Installed on Your System

Before attaching a new disk drive to your UNIX system, it is important to know how to determine, in a very quick and easy manner, what the currently installed logical device names of the disk drives actually attached and usable on your system are. What we mean by "attached and useable" is that the disk drive is properly connected and recognized by the system, and has a device driver that the system can use to communicate with it.

Before and after: If you want to find out the logical device name of a new disk you want to add to the system, use one of the following methods to see what disks are on your system before you add the new disk, and then use the same method after the new disk has been added and note the difference. The different or new logical name that appears will be the logical name of the new disk.

The simple methods that follow show how to determine what disk drives are attached and usable on your system, and what the logical device names of those and any others you might want to add to your system are. These methods are done differently in Solaris and PC-BSD.

PC-BSD methods: Change your current working directory to **/dev**. Type **`ls`**. Hard drives, for example, show up in the **`ls`** listing as **ada0**, **ada1**, etc. The full path to the first slice on one of these disks is specified as **/dev/ada0s1**. A USB bus device, like a thumb drive, would show up in the **`ls`** listing as **da0**, and the full path to the first slice on it would be **/dev/da0s1**.

Additionally, you can type **`gpart show`** on the command line in a terminal window, and it will list the drives available for slicing, such as **ada0**, **ada1s1**, or **da0s1** for disks and USB devices. The logical device names to all of those devices is provided in a compact and easily understood listing.

Solaris methods: For SCSI, SATA, or IDE bus hard disk drives, type **`format`** on the command line as superuser. The output of this command shows your main hard drive—for example, disk **0**—as **/dev/dsk/c0d0s0**. Type `exit` to leave the format facility.

For USB bus thumb drives or hard drives, type **`rmformat`** on the command line as superuser. The output of this command shows, for example, a USB thumb drive as **/dev/rdsk/c9t0d0p1**. A CD or DVD writer would show up here too, not on the USB bus, but on the SATA bus.

Additionally, you can use the Gparted Partition Editor, as shown in Example 23.3.

EXERCISE 23.9

Insert a USB thumb drive into either a PC-BSD or Solaris computer, and mount it if necessary. What is the logical device name for this thumb drive, and how did you find it out?

23.4.3 Adding a New Disk to the System

The following three examples show how to add a new hard disk to your system:

Example 23.3 uses a GUI method in Solaris. This GUI technique not only lets you find out what the logical device names of disks are, but also allows you to slice newly added disks at the same time. It uses the Gparted Partition Editor program that comes installed with Solaris.

Example 23.4 uses a GUI method in PC-BSD, by launching the PC-BSD Disk Manager. It also lets you find out the logical device names of disks on your system, and allows you to slice newly added disks.

Example 23.5 uses a combination of GUI and command line instructions in PC-BSD to allow you to do a postinstallation addition of an external USB hard drive with a ZFS file system to your computer.

In PC-BSD you can use the `gpart` command to do the same operations that the GUI Disk Manager does. The general form of the `gpart` command is as follows.

SYNTAX

`gpart sub-command [option(s)] [option argument(s)] [geom]`

Purpose: The **`gpart`** utility is used to partition GEOM providers, normally disks.

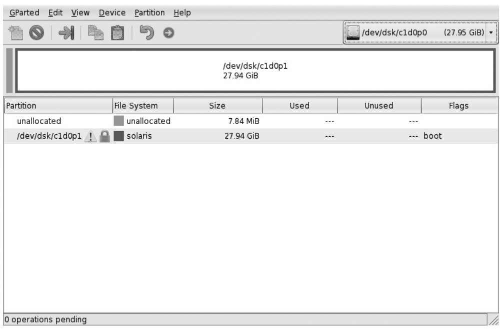
Commonly used options/features:

- **`gpart create -s GPT ada0`** Create a GPT scheme on **ada0.**
- **`gpart add -s 512M -t freebsd-zfs ada1`** Create a 512 MB freebsd-ufs partition to contain a ZFS file system from which the system can boot.
- **`gpart add -s 15G -t freebsd-ufs -a 4k da0`** Create a 15 GB freebsd-ufs partition to contain a ZFS file system and aligned on 4KB boundaries.
- **`gpart add -t freebsd-zfs -l zfs1 ada1`** Create a ZFS partition on all the free space on ada1.
- **`gpart show`** Show the status of disks on the system.
- **`gpart delete -i 1 da0`** Delete partition 1 on **da0.**
- **`gpart destroy -F da0`** Delete the partition table on **da0**.

You can use `gpart` to partition and format new disk drives you add to the system. See the man page for `gpart` on your system for more description of its subcommands, options, option arguments, and command arguments.

Example 23.3: Adding a New Disk and Using Gparted to Slice the Disk in Solaris

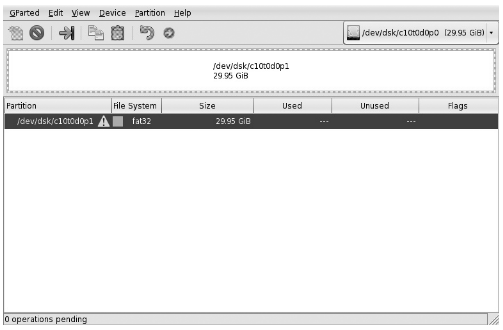
1. From the **Applications Menu>System Tools**, make the choice **Gparted Partition Editor**. If necessary, in a terminal window, type **`gparted`** as superuser.
2. The Gparted screen appears, as shown in Figure 23.6.



Partition	File System	Size	Used	Unused	Flags
unallocated	unallocated	7.84 MiB	---	---	
/dev/dsk/c1d0p1	solaris	27.94 GiB	---	---	boot

FIGURE 23.6 Gparted main screen display.

3. The current disks attached to the system appear in the menu bar at the upper right. Make note of all the complete paths to the current disks. For example, the boot or root disk might be designated as **/dev/dsk/c1d0p0**, as seen in Figure 23.6.
4. Power down the system if necessary, and add the new disk. Power the system back up.
5. Relaunch Gparted.
6. When the Gparted screen reappears, your new disk drive should show in the menu bar in the upper-right corner of the Gparted screen. If the disk drive you just added doesn't appear in the Gparted listing, you can't use that disk drive. If it does appear, continue to the next step.
7. Scroll in that bar until you reach the disk you just added to the system.
8. Pick that new disk in the menu bar. It is shown in the main Gparted pane. Click on that disk in the main Gparted pane. You can now partition and format that new disk. In our example it is shown as **/dev/dsk/c10t0d0p0**, a new disk we inserted at step 4. It has a FAT32 file system on it, and it is 29.95 GB in size. See Figure 23.7.
9. From the pull-down menus at the top of the Gparted window, make the menu choice **Partition>Delete**. This will delete the partition information on that disk. It is now a pending operation.
10. In order to execute the pending operation, make the pull-down menu choice **Edit>Apply All Operations**. In the warning window, click **`Apply`**. A window appears showing you the progress, and hopefully successful application, of the pending operation. Click close in that window when the operation is complete.
11. The new disk should now be unallocated. Click on its listing in the main Gparted pane. Make the pull-down menu choice **Device>Create Partition Table**. Click **`Create`** in the warning window. Everything on that disk will be erased! When Gparted has created a new partition table, click on that disk again in the main Gparted pane.
12. Make the pull-down menu choice **Partition>New**. The `Create New Partition` window appears on screen, as shown in Figure 23.8. The defaults



/dev/dsk/c10t0d0p0 (29.95 GiB)

/dev/dsk/c10t0d0p1
29.95 GiB

Partition	File System	Size	Used	Unused	Flags
/dev/dsk/c10t0d0p1 ⚠	▨ fat32	29.95 GiB	---	---	

0 operations pending

FIGURE 23.7 Partition information in Gparted window.

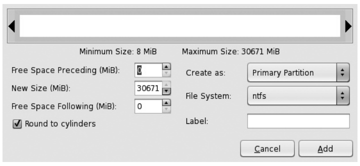
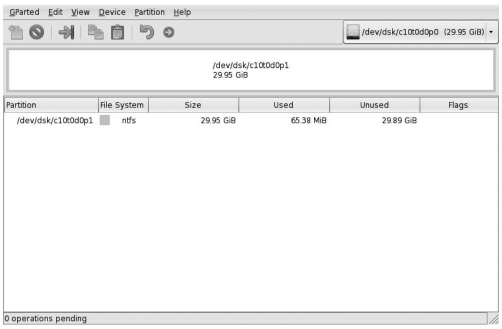

Minimum Size: 8 MiB
Maximum Size: 30671 MiB
Free Space Preceding (MiB): 0
New Size (MiB): 30671
Free Space Following (MiB): 0
[✓] Round to cylinders
Create as: Primary Partition
File System: ntfs
Label:

FIGURE 23.8 Gparted create new partition window.



/dev/dsk/c10t0d0p0 (29.95 GiB)

/dev/dsk/c10t0d0p1
29.95 GiB

Partition	File System	Size	Used	Unused	Flags
/dev/dsk/c10t0d0p1	ntfs	29.95 GiB	65.38 MiB	29.89 GiB	

0 operations pending

FIGURE 23.9 Gparted main window showing completed partition.

for the new partition as seen in Figure 23.8 are to take the whole disk up with this partition, create it as a primary partition (the only partition allowed in the ZFS file system), and set the file system as ntfs.

13. Add a label designation of your choice in the `Label` field. Leave all of the other defaults in place. Click the `Add` button. Make the pull-down menu choice **Edit>Apply All Operations**, and follow up on this choice as you did in step 10.
14. You now have a partitioned and formatted disk usable on the system, for our system shown in Figure 23.9.
15. Quit Gparted by making the pull-down menu choice **Gparted>Quit**.

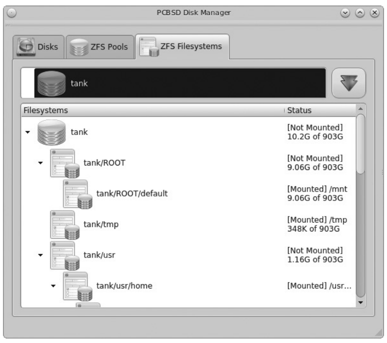
Example 23.4: Adding a New Disk and Partitioning It in PC-BSD

The following instructions assume you have only one hard disk on your system to begin with; on our system it is listed as **ada0**. You should deploy the methods shown in Section 23.4.2 to find out the logical name of the new disk. You can identify the original disk on your system when you reach step 4 because it has a zpool named **tank** already on it. The **tank** zpool is the default pool created when you install the system, and it contains the boot slice, swap space, and operating system.

These instructions work for adding SATA hard disks internally. USB external hard disks are covered in the next example. If the disk you are adding does NOT appear in the listing as a new disk in step 4, you cannot easily use it on your system without obtaining the device drivers for it.

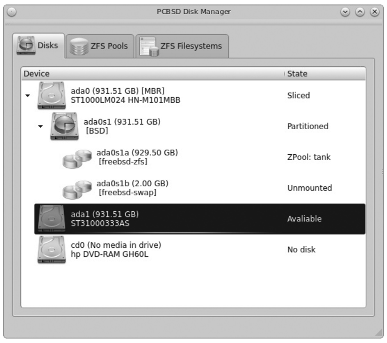
Before you begin this example, it is worth noting that you can also use the `gpart` command on the command line to accomplish the same operations shown. See the man page for `gpart` on your system for command options and arguments, as well as examples of how to partition and format a disk. `gpart` does allow formatting to freebsd-zfs. Also, knowing the logical device name of the disk you are adding is important, so that you can supply this logical device name as the `geom` argument to the `gpart` command.

1. Power down the system, and add the new disk drive. Power the system back up.
2. From the Kickoff Application Launcher menu on the extreme lower-left corner of the KDE4 desktop, make the choice **Applications>Utilities>Root Terminal**. Provide the root password when prompted.
3. A root terminal window opens on your screen. Type **`pc-zmanager`** in it. The PC-BSD Disk Manager screen opens, as shown in Figure 23.10.
4. Click on the **`Disks`** tab at the top of the PC-BSD Disk Manager window. The designation and listing for your newly added disk should appear near the bottom of the disks pane, similar to Figure 23.11. The designation display of the new disk on our system in that figure is `ada1 (931.51 GB) ST31000333AS`, and it is listed as `Avaliable` (*sic*). On our system the **cd0** disk was already there at installation, and so was **ada0** (the disk with a slice containing **tank**). Your new disk may have a different designation, depending on how many disks are on the system.



Filesystems	Status
tank	[Not Mounted] 10.2G of 903G
tank/ROOT	[Not Mounted] 9.06G of 903G
tank/ROOT/default	[Mounted] /mnt 9.06G of 903G
tank/tmp	[Mounted] /tmp 348K of 903G
tank/usr	[Not Mounted] 1.16G of 903G
tank/usr/home	[Mounted] /usr...

FIGURE 23.10 Disk Manager window.

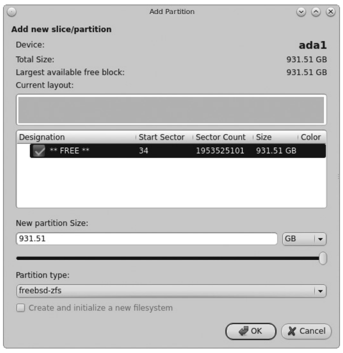


Device	State
ada0 (931.51 GB) [MBR] ST1000LM024 HN-M101MBB	Sliced
ada0s1 (931.51 GB) [BSD]	Partitioned
ada0s1a (929.50 GB) [freebsd-zfs]	ZPool: tank
ada0s1b (2.00 GB) [freebsd-swap]	Unmounted
ada1 (931.51 GB) ST31000333AS	Avaliable
cd0 (No media in drive) hp DVD-RAM GH60L	No disk

FIGURE 23.11 Disk Manager disks tab display.

5. To create a slice (partition) table on the disk, right-click on it and make the choice **Create GPT Partition Table**. The designation display of that disk is now followed by `[GPT]`. Anything on it is now erased.
6. Right-click on the new disk, and make the choice **Add a new slice**. The `Add Partition` window appears on screen, as shown in Figure 23.12. Only change the partition type to freebsd-zfs, and then click OK.
7. You have just created a new partition table and slice using the whole disk with a FreeBSD file system known as ZFS on it, and know the path to it in **/dev**. That path information is crucial to doing zpool creation. See Figure 23.13.

   You use the path when you are doing ZFS file system maintenance operations, such as creating zpools and file systems on those pools, and associating them with *virtual devices* (vdevs). This new disk is now a vdev on your system. Vdevs are explained in detail in Chapter 24.
8. You now have the choice of using the ZFS `Pools` tab or the ZFS `File systems` tab to graphically create and manipulate zpools and file systems on them. You can also use ZFS commands typed at the command line to achieve everything available in the Disk Manager, and substantially more as well. Both facilities are interoperable and designed to fit the style of interaction you are most comfortable with.



Device: ada1
Total Size: 931.51 GB
Largest available free block: 931.51 GB
Current layout:

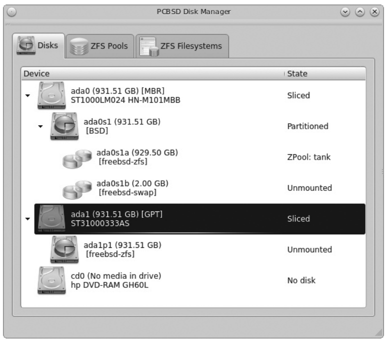
Designation	Start Sector	Sector Count	Size	Color
** FREE **	34	1953525101	931.51 GB	

New partition Size: 931.51 GB

Partition type: freebsd-zfs

Create and initialize a new filesystem

FIGURE 23.12 Disk Manager add partition display.



Device	State
ada0 (931.51 GB) [MBR] ST1000LM024 HN-M101MBB	Sliced
ada0s1 (931.51 GB) [BSD]	Partitioned
ada0s1a (929.50 GB) [freebsd-zfs]	ZPool: tank
ada0s1b (2.00 GB) [freebsd-swap]	Unmounted
ada1 (931.51 GB) [GPT] ST31000333AS	Sliced
ada1p1 (931.51 GB) [freebsd-zfs]	Unmounted
cd0 (No media in drive) hp DVD-RAM GH60L	No disk

FIGURE 23.13 Disk Manager with added partition.

9. When ready, you can quit the PC-BSD Disk Manager by clicking on the destroy button in the upper right-hand corner of the window, and you can also exit from the root terminal you launched in step 2.

Example 23.5: Adding an External USB Hard Drive and Partitioning It in PC-BSD

A common postinstallation addition to a PC-BSD system is of an external USB hard drive that needs to be formatted and partitioned before it can be mounted and used on the system.

The following instructions assume you already have an internally mounted hard disk on your system to begin with; on our system it is listed as **ada0**. You should deploy the methods shown in Section 23.4.2 to find out the logical name of the new disk you want to add. USB drives usually are designated as **daXpY**, where **X** is a number starting at 0, and **Y** is a number starting at 1.

The instructions work for adding external USB hard disks, SATA or IDE, usually mounted in a powered enclosure. They also show how to detach that drive from the system. After the first step, they also assume that it is partitioned and formatted in such a way as to be incompatible with the PC-BSD system. If the disk you are adding does not appear in the **/dev** listing as a new disk you cannot easily use it on your system without obtaining the device drivers for it.

Also, from the KDE4 Kickoff Applications Launcher, you can make the System Settings menu choice to affect the automounting process of USB removable media. One of the System Settings icons is `Removable Devices`. When you click on this icon, you can turn automounting on or off, and also configure other settings for removable media. By default, automounting is off. The Mount Tray, a KDE4 desktop icon, is shown in the Panel as item F in Table 22.4, and illustrated in Chapter 22, Figure 22.12, "KDE4 Panel components." It allows for easy, GUI-based management of removable media that you can add to or remove from the system.

1. Power on the external hard drive enclosure, and connect the USB cable from the enclosure to a USB port on the computer. At this point you could determine the logical device name of the newly added external USB drive using the methods shown in Section 23.5.2. But we show how to do this in the following steps using Disk Manager.

   The Mount Tray should notify you that it has been discovered but it is not mounted. At this point you could attempt to mount it from the Mount Tray, and if it has been properly partitioned and formatted, it may mount and be usable. Game over!

   It is important to realize at this point that if you are going to transfer files to or from this external drive, it should be usable and work for that purpose. You can use the PC-BSD File Manager to accomplish transferring files to and from this USB drive. But if the Mount Tray recognizes it, and when you try to mount it you get an error like `Could not mount device devicename on /root/media/devicename`, proceed to step 2.

2. Launch a command line console window and become the superuser.
3. In the console window, type **`pc-zmanager`** and press <**`Enter`**>. The PC-BSD Disk Manager screen opens.
4. Click on the **`Disks`** tab at the top of the PCBSD Disk Manager window. The designation and listing for your newly added disk should appear near the bottom of the disks pane.

   The designation display of the logical device name for the new external USB hard disk on our system is `da0 (38.29 GB), Maxtor 6 E040L0` and it is listed as `Sliced` (meaning partitioned). It also is shown in the `Disks` pane of the Disk Manager as having a single partition already on it, named **da0s1**. Your new disk may have a different designation, depending on how many disks are on the system.
5. This step allows you to destroy all partitions and the partition table on the disk, and allows you to create a GPT partition table on it. Right-click on any of the partitions already existing and make the **Destroy this slice** choice for all of them. When all existing partitions are destroyed, right-click on **da0**, and make the choice to **Delete partition table**. Then, right-click on **da0** and make the choice **Create GPT partition table**. The designation display of that disk is now followed by `[GPT]`. Anything on it is now erased. It is shown as `Avaliable` (*sic*).
6. Right-click on the new disk, and make the choice **Add a new slice**. The `Add Partition` window appears on screen. Change the partition type to freebsd-zfs, and put a check mark in the box that is labeled `Create` and initialize a new file system. The new partition on our system showed in the `Disks` pane of the Disk Manager as **da0p1**. Then click OK. You can now quit the Disk Manager by using the destroy button in the upper-right corner of the window.

   It is also possible to change the size of the partition, and add multiple partitions to this drive if you want to from the `Add Partition` window. For our purposes here, we only showed how to create a single primary partition that took up the whole disk.
7. You can now create ZFS zpools and ZFS datasets on this disk with the **`zpool create`** and **`zfs create`** commands. For example, to create a ZFS zpool on it named testing, type:

   ```
   #zpool create testing /dev/da0p1
   ```
8. To own the files you copy to and from the new external USB drive, you may need to use the **`chown`** and **`chgrp`** commands on the directory that a **`zpool create`** command mounts the file system to.
9. You have just created, on an external USB hard drive, a new partition table and slice using the whole disk with a ZFS file system on it. You also know the logical device name of it in **/dev**, and know its mount point on the system after you create a ZFS zpool on it.

10. To detach the USB external hard drive, and destroy the zpool, partition, and data on it, destroy the zpool on it using the command **`zpool destroy testing`**, and repeat step 5 on it. Then you can safely remove the USB external hard drive from the computer.

EXERCISE 23.10

1. Use the `gpart` command in PC-BSD to partition with a primary partition, format to freebsd-zfs, and add a new internal SATA bus hard disk to your system.
2. If you haven't done so already, use the GUI methods of Example 23.5, to add an external USB hard drive to your PC-BSD system.
3. What `gpart` commands would need to be executed to accomplish the same things that the Disk Manager GUI does in adding an external USB hard disk, according to the methods of Example 23.5? To test those commands in sequence, redo Example 23.5 using the `gpart` command only. The

In place of informational step 7, create a zpool on the external USB drive using the command:

```
zpool create test7 /dev/da0p1
```

The USB external hard drive will be mounted at **/test7**. That is the pathname to where you can copy files to and from the USB external hard drive.

23.5 ADDING A PRINTER TO THE SYSTEM

A printer is a common peripheral addition to any computer system.

PC-BSD uses the *Common UNIX Printing System* (CUPS) to handle printing. The PC-BSD Control Panel provides a graphical front-end for adding and managing printers. You can launch the PC-BSD Control Panel from its Desktop icon, or as superuser by typing `pc-controlpanel` in a console or terminal window. In the Control Panel, printer control is found in the **Hardware** group, and it is named **Printing**.

While the Control Panel graphical utility is easy to use, it may or may not automatically detect your printer, depending upon how well your printer is supported by an open-source print driver. Your printer may be found quickly and easily, along with its driver, thus allowing you to breeze through the configuration operations. If your printer configuration does not work, read the following section for advice on how to locate the correct driver for your printer.

23.5.1 Researching Your Printer

Before configuring your printer, you have to find out if a print driver exists for your particular model, and if so, which driver is recommended. If you are planning to purchase a printer, this is definitely good information to know beforehand. You can look up the vendor and model of the printer in the Open Printing Database, found at www.openprinting.

org/printers, which will indicate if the model is supported and if there are any known problem with the print driver. We found the HP LaserJet 1200 in the database, and the HPLIP driver was recommended for it.

In PC-BSD, the HPLIP driver is available as an optional system component called **pcbsd-meta-hplip**. You can see if the driver is installed, and install it if it is not, using the AppCafe. You launch the AppCafe as superuser by typing `pc-softwaremanager` in a console or terminal window.

In order to see the driver, make sure that the `Raw Packages` box is checked in the pull-down **Browser View** menu. In the main AppCafe window, make the **Browse Categorie**s choice. Then, under the **Miscellaneous** category, find the **pcbsd-meta-hplip** driver. Once you have located the driver in AppCafe, install it.

23.5.2 Adding a Printer

The most efficient way to add a printer to your printer to your PC-BSD system is to follow the instructions in the PC-BSD handbook for your version of the software.

EXERCISE 23.11

Add a printer to your PC-BSD system.

23.5.3 Adding a Printer to Solaris

The most efficient way to add a printer to Solaris is to consult the online documentation for your version of the Solaris software.

EXERCISE 23.12

Add a printer to your Solaris system.

23.6 FILE SYSTEM BACKUP STRATEGIES AND TECHNIQUES

The general necessity of backing up user and system files on your UNIX system as a part of system administration and maintenance should seem obvious to anyone responsible for the administration of the system. And it should also be obvious to an ordinary user, if they want to maintain the integrity of her important personal files on a desktop or laptop in a home computer use case.

According to UNIX system professionals, there is an easy-to-remember and important set of considerations you must make when backing up the system as the system administrator. This set of considerations can be posed in simple question form as: *how*, *what*, *why*, *when*, *where*, and *who*? Some of the answers to these simple questions can be dovetailed together, and we repeat here the selected list of example answers as follows:

"How," most importantly, means the command, utility, application, or combination of hardware and software used to accomplish the backup and archive. These facilities are described in the subsections that follow. It also means backing up/archiving incrementally, in a rolling fashion for example, or using various strategies.

"What" refers to whether you back up just some of the user files and user account files, all of them, only certain kinds of documents, the whole disk drive, multiple disk drives, all or a selected subset of the system files, etc.

"Why" means deciding on the relative importance of "what" you are backing up.

"When" means every time you save a particular file, hourly, once a day, once a week, once a month, and at what time of day exactly, such as 3:00 a.m.

"Where" means saving to a local disk, to Dropbox/Google/Amazon, to a USB thumb drive manually, to another computer or NAS on your home network, automatically by cron, to another hard disk manually, totally, incrementally, to RAID of various levels, or any variant or combination of these.

"Who" refers to the initiator or executor, such as by cron automatically, by the designated system administrator either manually or semiautomatically, on Dropbox, Google, or Amazon.

23.6.1 A Strategic Synopsis and Overview of File Backup Facilities

There are several strategies that a system administrator can use in confidently and efficiently backing up the system and user file components of UNIX . Table 23.7 gives a basic overview of those strategies and the facilities that implement them on a modern UNIX system. In the sections that follow the table, we briefly give details of these facilities.

23.6.2 `tar` and `gtar`

The UNIX operating system has several utilities that allow you to archive your files and directories in a single file, and `tar` is the most popular, widely used, and traditional command that allows you to do this.

TABLE 23.7 `tar` Facilities

Backup Facility	Description
`tar, gtar`	Command and options to pack a file or a directory hierarchy as an ordinary disk file for backup, archiving, or moving to another location or system. The GNU version is `gtar`.
`cpio`	Less popular than `tar`, but with much of the same functionality.
`rsync`	A disk space-efficient command to copy files and directories.
`dump/restore`	PC-BSD incremental file backup and restore.
`dd`	A simple and abbreviated backup utility.
`ufsdump/ufsrestore`	Solaris version of incremental file dump.
`zfs snapshot`	Built-in commands and options in ZFS that offer a variety of backup modes.
`Script files`	Administrator or user-written shell scripts or other programming language backup systems that can use all of these commands in them.
`Third-party software`	Many products, both local and online. The most significant for ordinary use are Clonezilla and FileZilla.

The `tar` (short for *tape archive*) utility was originally designed to save file systems on tape as a backup so that files could be recovered in the event of a system crash. It is still used for that purpose, but it is also commonly used now to pack a directory hierarchy as an ordinary disk file. Doing so saves disk space and transmission time while a directory hierarchy is being transmitted electronically. The saving in disk space results primarily from the fact that empty space within a cluster is not wasted. A brief description of the `tar` utility follows.

Additionally, `gtar` (the GNU version of `tar`) has some important functional features not included in `tar`. System administrators normally use a cost-effective archival medium for archiving complete file system structures as backups so that, when a system crashes for some reason, files can be recovered. UNIX-based computer systems normally crash for reasons beyond the operating system's control, such as a disk head crash because of a power surge. UNIX rarely causes a system to crash because it is a well-designed, coded, and tested operating system. In a typical installation, backup is done every day during off hours (late night or early morning) when the system is not normally in use.

The general structure of the `tar` command is as follows.

SYNTAX

`tar [options] ['device'] ['pattern']`

Purpose: Archive (copy in a particular format) files to or restore files from an archival medium (which can be an ordinary file). Directories are archived and restored recursively.

Options and Option Arguments:

Option Format: **`Function _ letter [Modifier]`**

`Function _ letter:`

- **`c`** Create a new tape and record archive **files** on it
- **`r`** Record **files** at the end of **tape** (append operation); list **tape**'s contents (names of files archived on it) in a format such as **`ls -l`**; update **tape** by adding **files** on it if not on or if modified since last written to **tape**
- **`x`** Extract (restore) **files** from **tape**; entire **tape** if none specified

`Modifier:`

- **`b N`** Use N as the blocking factor (1 default; 20 maximum).
- **`f 'Arch'`** Use **`'Arch'`** as the archive for archiving or restoring files; default is **/dev/mto**. If **`'Arch'`** is **-**, standard input is read (for extracting files) or standard output is written (for creating an archive)—a feature used when **`tar`** is used in a pipeline.
- **`h`** Follow symbolic links.
- **`l`** Display error message if links are not found.
- **`o`** Change ownership (user ID and group ID) to the user running **`tar`**.
- **`v`** Use verbose mode: Display function letter **`x`** for extraction or for an archive.

Command Arguments:

`'device'`: A destination for the archive, which can be a filename, a pathname, or a remote location

`'pattern'`: The source to be archived, based on filename expansion syntax; can be a file, files, or directories

23.6.2.1 Archiving and Restoring Files Using `tar`

A normal UNIX user can archive their work if they want to. They would normally need to do this to archive files related to a project so that they can transfer them to someone via e-mail, ftp, ssh, or via secondary storage media (USB thumb drive, DVD, or CD-ROM). The primary reason for making an archive is the convenience of dealing with (sending or receiving) a single file instead of a complete directory hierarchy. Without an archive, the sender might have to send several files and directories (a file structure) that the receiver would have to restore in their correct hierarchical structure. Without an archiving facility, depending on the size of the files and directory structure, the task of sending, receiving, and reconstructing the file structure can be very time consuming.

In Chapter 7, we discussed file compression by using the `compress` and `pack` commands and pointed out that compression saves disk space and transmission time. However, compressing small files normally does not result in much compression. Moreover, compressing files of one cluster in size (the minimum unit of disk storage; one or more sectors) or less does not help save disk space even if compression does result in smaller files, because the system ends up using one cluster to save the compressed file anyway. But if compression does result in a smaller file, you do save time in transmitting the compressed version. If the disk block size is 512 bytes and a cluster consists of more than two blocks, you can use the `tar` command to pack files together in one file, with a 512-byte tar header at the beginning of each file, as shown in Figure 23.14.

23.6.2.2 Archiving Files

You can use the `tar` command for archiving (also known as packing) a list of files and/or directories by using the `c` or `r` option. The `c` option creates a new archive, whereas the `r` option appends files at the end of the current archive. The most common use of the `tar` command is with the `c` option.

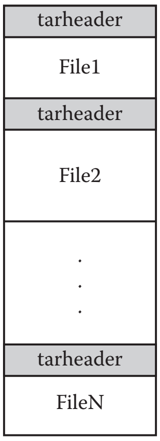


FIGURE 23.14 Tar disk block.

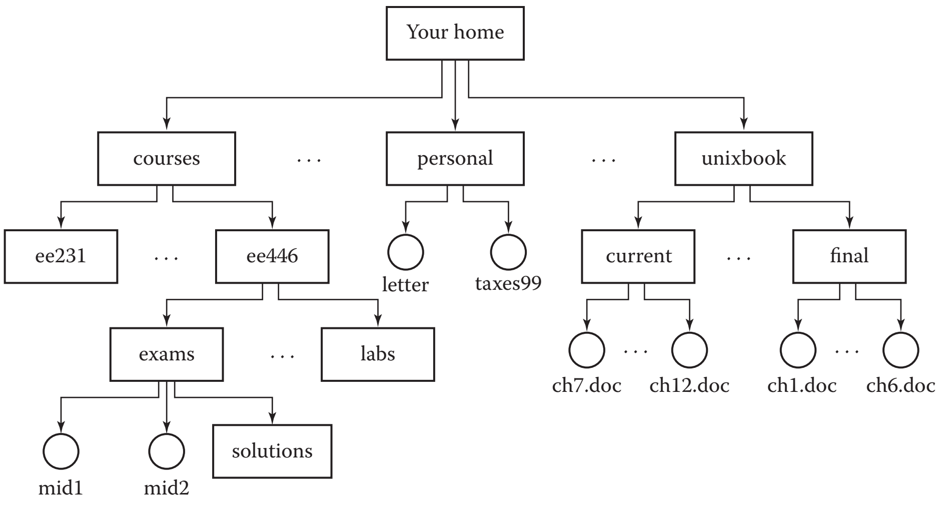


FIGURE 23.15 Directory structure.

In the examples presented in this chapter, we use the directory structure shown in Figure 23.15. The following session shows that there are two directories under the **unixbook** directory called **current** and **final**. In addition, each of these directories contains six files (see Figure 23.15), displayed by the `ls -l` command.

```
$ cd unixbook
$ pwd
/users/sarwar/unixbook
% ls -l
drwx------   2     sarwar   512       Jul 22      13:21 current
drwx------   2     sarwar   512       Jul 22      13:21 final
$ cd current
$ ls -l
-rw-------   1     sarwar   204288    Jul 19      13:06 ch07.doc
-rw-------   1     sarwar   87552     Jul 19      13:06 ch08.doc
-rw-------   1     sarwar   86016     Jul 19      13:06 ch09.doc
-rw-------   1     sarwar   121344    Jul 19      13:06 ch10.doc
-rw-------   1     sarwar   152576    Jul 19      13:06 ch11.doc
-rw-------   1     sarwar   347648    Jul 19      13:06 ch12.doc
$ cd ..
$ cd final
$ ls -l
-rw-------   1     sarwar   41984     Jul 19      13:06 ch1.doc
-rw-------   1     sarwar   54272     Jul 19      13:06 ch2.doc
-rw-------   1     sarwar   142848    Jul 19      13:06 ch3.doc
-rw-------   1     sarwar   86528     Jul 19      13:06 ch4.doc
-rw-------   1     sarwar   396288    Jul 19      13:06 ch5.doc
```

```
-rw-------  1     sarwar   334848   Jul 19      13:06 ch6.doc
$
```

If you want to create an archive of the unixbook directory on a tape drive **/dev/rmt/t0** (the name might be different on your system), you can use the following command after changing the directory to **unixbook**. The `v` (verbose) option is used to view the files and directories that are being archived. Unless you are the system administrator, in all likelihood you do not have access permission to use (read or write) the tape drive. Thus, the shell will give you the following error message:

```
$ tar cvf /dev/rmt/t0 .
tar: /dev/rmt/t0: Permission denied
$
```

However, if you do have proper access permissions for the tape drive, an archive of the **unixbook** directory is created on the tape. Not only do you need access privileges to **/dev/rmt/t0**, but you also need to mount it first.

However, you can also create an archive on a disk file—in a directory that you have the write permission for—by using the following command. Here we made **~/unixbook** our current directory and created a *tar archive* of this directory in a file called **unixbook.tar**. Note that **.tar** is not an extension required by the `tar` utility. We have used this extension because it allows us to identify tar archives by looking at the file name. With no such extension, we have to use the file command to identify our tar archive files, as shown in the last command line in the session (in case you need reminding what the file command does).

```
$ cd ~/unixbook
$ tar cvf unixbook.tar.
tar: ./unixbook.tar same as archive file
a ./final/0K
a ./final/ch1.doc 41K
a ./final/ch2.doc 53K
a ./final/ch3.doc 140K
a ./final/ch4.doc 85K
a ./final/ch5.doc 387K
a ./final/ch6.doc 327K
a ./current/0K
a ./current/ch07.doc 200K
a ./current/ch08.doc 86K
a ./current/ch09.doc 84K
a ./current/ch10.doc 119K
a ./current/ch11.doc 149K
a ./current/ch12.doc 340K
$ ls -l
```

```
drwx------  2    sarwar  512 Jul   22    13:21 current
drwx------  2    sarwar  512 Jul   22    13:21 final
-rw-------  1    sarwar  2064896   Jul   22    13:47 unixbook.tar
$ file unixbook.tar
unixbook.tar:           USTAR tar archive
$
```

You can also create the tar archive of the current directory by using the following command. The - argument informs `tar` that the archive is to be sent to standard output, which is redirected to the **unixbook.tar** file. As we discussed in Chapter 12, the back quotes (grave accents) are used for command substitution—that is, to execute the find command and substitute its output for the command, including the back quotes. The output of the `find . -print` command (the names of all the files and directories in the current directory) are passed to the `tar` command as its parameters. These file and directory names are taken as the list of files to be archived by the `tar` command. Thus, the net effect of the command line is that a tar archive of the current directory is created in **unixbook.tar**.

```
$ tar cvf - 'find . -print' > unixbook.tar
$
```

In the following in-chapter exercise, you will use the `tar` command with the `c` option to create an archive of a directory.

EXERCISE 23.13

Create a tar archive of a subdirectory hierarchy of your choosing under your own home directory on your UNIX system. What command line(s) did you use? What is the name of your archive file?

23.6.2.3 Restoring Archived Files

You can restore (unpack) an archive by using the function option `x` of the `tar` command. To restore the archive created in Section 23.6.2.2 and place it in a directory called **~/backups**, you can run the following command sequence. The `cp` command copies the archive file, assumed to be in your home directory, to the directory (**~/backups**) where the archived files are to be restored. The `cd` command is used to make the destination directory the current directory. Finally, the `tar` command is used to do the restoration. Notice that the destination directory is the current directory.

```
$ cp unixbook.tar ~/backups
$ cd ~/backups
$ tar xvf unixbook.tar
x ., 0 bytes, 0 tape blocks
```

```
x ./final, 0 bytes, 0 tape blocks
x ./final/ch1.doc, 41984 bytes, 82 tape blocks
x ./final/ch2.doc, 54272 bytes, 106 tape blocks
x ./final/ch3.doc, 142848 bytes, 279 tape blocks
x ./final/ch4.doc, 86528 bytes, 169 tape blocks
x ./final/ch5.doc, 396288 bytes, 774 tape blocks
x ./final/ch6.doc, 334848 bytes, 654 tape blocks
x ./current, 0 bytes, 0 tape blocks
x ./current/ch07.doc, 204288 bytes, 399 tape blocks
x ./current/ch08.doc, 87552 bytes, 171 tape blocks
x ./current/ch09.doc, 86016 bytes, 168 tape blocks
x ./current/ch10.doc, 121344 bytes, 237 tape blocks
x ./current/ch11.doc, 152576 bytes, 298 tape blocks x ./current/
ch12.doc, 347648 bytes, 679 tape blocks
$ ls -l
drwx------  2     sarwar      512        Jul 22 13:21 current
drwx------  2     sarwar      512        Jul 22 13:21 final
-rw-------  1     sarwar      2064896    Jul 22 13:47 unixbook.tar
$
```

The **unixbook.tar** file remains intact after it has been unpacked. This result makes sense considering that the primary purpose of the tar archive is to back up files, and it should remain intact after restoration in case the system crashes after the file is restored but before it is archived again. After restoration of the **unixbook.tar** file, your directory structure looks like that shown in Figure 23.16.

At times, you might need to restore a subset of files in a tar archive. System administrators often do this after a system crashes (usually caused by a disk head crash resulting from a power surge) and destroys some user files with it. In such cases, system administrators restore only those files from the tape archive that reside on the damaged portion of the disk. Selective restoration is possible with the function option `x` as long as the pathnames of the files to be restored are known. If you do not remember the pathnames of the files to be restored, you can use the function option `t` to display the pathnames of files and directories on the archive file. The output of the `tar` command with the `t` option is in a format similar to the output of the `ls -l` command, as shown in the following session. As marked in the sample, the first field specifies file type and access

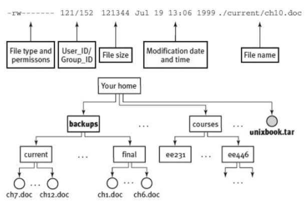


FIGURE 23.16 Restored directory structure.

permissions, the second field specifies user ID/group ID of the owner of the file, the third field shows the file size in bytes, the next several fields show the time and date that the file was last modified, and the last field shows the pathname of the file as stored in the archive.

If an archive contains a large number of files, you can pipe the output of the `tar t` command to the more command for page-by-page view. If you know the name of the file but not its pathname, you can pipe output of the `tar` command with the `t` option to the `grep` command. Files can also be restored, or their pathnames viewed, selectively. The following session illustrates these points.

```
$ tar -tvf unixbook.tar
drwx------  121/152     0     Jul   22    13:47 2015  ./
drwx------  121/152     0     Jul   22    13:21 2015  ./final/
-rw-------  121/152     41984 Jul   19    13:06 2015  ./final/ch1.
doc
-rw-------  121/152     54272 Jul   19    13:06 2015  ./final/ch2.
doc
-rw-------  121/152     142848Jul   19    13:06 2015  ./final/ch3.
doc
-rw-------  121/152     86528 Jul   19    13:06 2015  ./final/ch4.
doc
-rw-------  121/152     396288Jul   19    13:06 2015  ./final/ch5.
doc
-rw-------  121/152     334848Jul   19    13:06 2015  ./final/ch6.
doc
drwx------  121/152     0     Jul   22    13:21 2015  ./current/
-rw-------  121/152     204288Jul   19    13:06 2015  ./current/
ch07.doc
-rw-------  121/152     87552 Jul   19    13:06 2015  ./current/
ch08.doc
-rw-------  121/152     86016 Jul   19    13:06 2015  ./current/
ch09.doc
-rw-------  121/152     121344Jul   19    13:06 2015  ./current/
ch10.doc
-rw-------  121/152     152576Jul   19    13:06 2015  ./current/
ch11.doc
-rw-------  121/152     347648Jul   19    13:06 2015  ./current/
ch12.doc
$ tar -tvf unixbook.tar | grep ch10.doc
-rw-------  121/152     121344      Jul 19      13:06 2015  ./
current/ch10.doc
$
```

If you want to restore the file **ch10.doc** in the **~/unixbook/current** directory, you can use the following command sequence. Be sure that you give the pathname of the file to be restored, not just its name.

```
$ cd ~/unixbook
$ tar -xvf ~/backups/unixbook.tar current/ch10.doc
$
```

The output of this `tar` command shows that the file **ch10.doc** has been restored in the **~/backups/current** directory. You can confirm this result by using the `ls -l ~/backups/ current` command.

In the following in-chapter exercises, you will use the `tar` command with `t` and `x` options to appreciate how attributes of the archived files can be viewed and how archived files can be restored.

EXERCISE 23.14

List the attributes of the files in the archive that you created in Exercise 23.13 and identify the sizes in bytes of all the files in it. What command did you use to do this?

EXERCISE 23.15

Copy the archive file that you created in Exercise 23.13 to a file called **mytar**. Unarchive it in a directory called **dir.backup** under your home directory. Show the commands you used for this task.

23.6.2.4 Copying Directory Hierarchies

You can use the `tar` command to copy one directory to another directory. You can also use the `cp -r` command to do so, but the disadvantage of using this command is that the file access permissions and file modification times are not preserved. The access permissions of the copied files and directories are determined by the value of umask, and the modification time is set to current time. Also, the `-r` option is not available on all UNIX systems.

Additionally, you can also use the `cp –p` command to preserve file permissions.

More commonly, the `tar` command is used to archive the source directory, create the destination directory, and *untar* (unpack) the archived directory in this latter directory. The entire task can be performed with one command by using command grouping and piping. In the following session, the **~/unixbook/examples** directory is copied to the **~/unixbook/ examples.bak** directory. The `tar` command to the left of pipe sends the archive to **stdout**, and the `tar` command to the right of pipe unpacks the archive it receives at its **stdin**.

```
$ (cd ~/unixbook/examples; tar -cvf - .) | (cd ~/unixbook/
examples.bak; tar -xvf -)
a ./ 0K
a ./Bshell/Domain, 14 bytes, 1 tape blocks a ./Bshell/IP, 18
bytes, 1 tape blocks
a ./Bshell/dns_demo1, 227 bytes, 1 tape blocks
...
a ./Bshell/fs.csh, 1531 bytes, 3 tape blocks a ./Bshell/dir1, 0
bytes, 0 tape blocks
```

```
a ./Bshell/copy, 2222 bytes, 5 tape blocks x ., 0 bytes, 0 tape
blocks
x ./Bshell/Domain, 14 bytes, 1 tape blocks x ./Bshell/IP, 18
bytes, 1 tape blocks
x ./Bshell/dns_demo1, 227 bytes, 1 tape blocks
...
x ./Bshell/fs.csh, 1531 bytes, 3 tape blocks x ./Bshell/dir1,
0 bytes, 0 tape blocks
x ./Bshell/copy, 2222 bytes, 5 tape blocks
$
```

The advantages of using this command line are that both `cd` and `tar` commands are available on all UNIX systems and that the copied files have the access permissions and file modification times for the source files.

You can also use the `tar` command to copy directories to a remote machine on a network. In the following command line, the **~/unixbook/examples** directory is copied to the **~/unix- book/examples.bak** directory on a remote machine called **upsun21**. The `rsh` command (see Chapter 11) is used to execute the quoted command group on **upsun21**. Because the `tar` command is not run in verbose mode, it runs silently and you do not see any output on the display screen.

```
$ (cd ~/unixbook/examples; tar cf - .) | rsh upsun21 "cd
~/unixbook/example.bak; tar xf -"
$
```

23.6.2.5 Software Distributions in the `tar` *Format and* `gtar`

Companies often use the `tar` command to distribute their software because it results in a single file that the customer needs to copy, and it saves disk space compared to the unarchived directory hierarchies that might contain the software to be distributed. Also, most companies keep their distribution packs (in the *tar* format) on their Internet sites, where their customers can download them via the `ftp` command. Thus, the tar format also results in less "copying" time and reduced work by the customer, who uses only one `get` (or `mget`) command (an ftp command) versus several sequences of commands if directory hierarchies have to be downloaded.

Because the sizes of software packages are large due to their graphical interfaces and multimedia formats, archives are compressed before they are put on ftp sites. The users of the software need to uncompress the downloaded files before restoring them.

Now consider a file, **tcsh-6.06.tar.Z**, that we downloaded from an ftp site. In order to restore this file, we have to uncompress and untar it, as shown in the following command sequence.

```
$ uncompress tcsh-6.06.tar.Z
$ tar xvf tcsh-6.06.tar
... [Output truncated]
$
```

If a software pack is distributed on a secondary storage medium (USB thumb drive, DVD, or CD- ROM), you need to copy the appropriate files to the appropriate directory and repeat these commands.

If you want to distribute some software that is stored in a directory hierarchy, you first need to make an archive for it by using the `tar` command and then compress it by using the `compress` command (or some other similar utility). These steps create a tar archive in a compressed file that can be placed in an ftp repository or on a Web page or sent as an e-mail attachment.

<u>Disk file backups are the most modern and cost-effective way to create archives and backups.</u>

The `-z` option of the GNU version of `tar`, `gtar`, can be used to generate the compressed version of the tar archive. This option can be used to restore the compressed version of the tar archive. The use of this option eliminates the use of the `gzip` (or `gunzip`) utility to compress or uncompress an archive and then the `tar` command; the two-step process can be performed by the `tar` command alone. In the following session, the first `tar` command generates a compressed tar archive of the current directory in the **~/unixbook/backups/ub.tar.gz** file, and the second `tar` command restores the compressed tar archive from the same file into the current directory.

```
$ gtar -zcf ~/unixbook/backups/ub.tar.gz .
$ gtar -zxvf ~/unixbook/backups/ub.tar.gz
... [Output truncated]
$
```

23.6.3 Other UNIX Archiving and Backup Facilities

In addition to the traditional tar facility, there are several other facilities and methods a system administrator can use to archive and backup individual user accounts, files, file systems, and the entire system itself. As previously stated, to get a more complete listing of the capabilities and options available for the command line facilities, consult the man pages on your system for these commands. We will briefly describe and give a simple example of some of the more modern and useful of these facilities and methods. Also, we provide a nontraditional methodology for doing archiving and backups in ZFS in Chapter 24.

23.6.3.1 cpio

As universally available as `tar` on UNIX systems, `cpio` allows the system administrator the ability to backup the entire system and transfer files between file systems. It may be used in conjunction with the `find` command, but not necessarily if you are backing up an entire file system. Its general format is `cpio –o  [aBcv]` for creating an archive, and `cpio –I  [Btv]  [pattern]` for restoring an archive.

For example, the following commands backup the **/home** directory to a USB thumb drive generically named **device**:

```
$cd /home
$touch level.1.cpio.timestamp
```

```
$find . -newer level.0.cpio.timestamp -print \ | cpio -oacvB >
device
…
$
```

23.6.3.2 rsync

As introduced in Chapter 16, the `rsync` command is a very space-efficient way to backup files and directories, particularly from one machine to another using ssh across a network. Its operation can also be automated by designing and implementing an appropriate entry in the **crontab** file. The general syntax of `rsync` for local or remote copying is

Local:

```
rsync [option(s)...] src... [dest]
```

Remote:

```
Pull(copying to a remote location): rsync [option(s)...] [user@]
host:src... [dest]
Push(copying from a remote location): rsync [option(s)...] src...
[user@]host:dest
```

where `option(s)` are the valid options for the `rsync` command, `src` is the source file or directory, and `dest` is the destination path.

An example of using `rsync` to copy a file named **rsynctest** from the current working directory to the local destination **/usr/home/bob/USBint** is

```
$ rsync -av rsynctest /usr/home/bob/USBint
sending incremental file list
rsynctest
sent 192 bytes received 35 bytes 454.00 bytes/sec
total size is 92 speedup is 0.41
$
```

An example of using `rsync` to copy an entire directory named **syncdir** locally is

```
$ rsync -av syncdir /usr/home/bob/USBint
sending incremental file list
syncdir/
syncdir/Chap16.doc
syncdir/backup1.py
syncdir/ossystem.py
sent 280,176 bytes received 77 bytes 186,835.33 bytes/sec
total size is 279,855 speedup is 1.00
0
$
```

An example of copying an entire directory in push mode remotely is:

```
$ rsync -av -e ssh syncdir2 bob@192.168.0.7:/Users/b/unix3e
Password:
building file list ... done
syncdir2/
syncdir2/Chap16.doc
syncdir2/backup1.py
syncdir2/ossystem.py
sent 280,171 bytes received 96 bytes 16,985.88 bytes/sec
total size is 279,855 speedup is 1.00
0
$
```

23.6.3.3 `dump` *and* `restore` *(PC-BSD)*

The `dump` and `restore` facilities are a simple set of complementary commands that have a small number of options, and are primarily used to back up the system. What differentiates `dump` and `restore` from facilities like `cpio` and `tar` is that `dump` writes a table of contents at the beginning of each volume so that you can selectively restore files from the archive. For the creation of a backup, the `dump` command's basic syntax is as follows:

```
dump options file blocking-factor density size device filesystem
```

where `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `a`, `c`, `L`, `n`, `r`, `R`, `S`, and `u` are various options, `file`, `blocking-factor`, `density`, and `size` are option arguments that must be supplied in the same order as their respective options given on the command line, and `device` and `filesystem` are the command arguments.

The `restore` command's basic syntax is as follows.

```
restore options blocking-factor file-number device
```

for restoration of a backup, where `options` are various options, `blocking-factor` and `file-number` are option arguments that must be supplied in the same order as their respective options given on the command line, and `device` is the command argument.

An example of how to use `dump` to create a full backup of **/home** to a tape drive named **/dev/rmt/t0** is

```
dump 1unbdsf 512 100000 10000 /dev/rmt/t0 /home
```

where the level is 1, the blocking factor `b` is 512 bytes, the volume density and size, `d` and `s`, are 100000 and 10000, the device to backup to is **/dev/rmt/t0**, and the directory to backup is **/home**.

An example of how to use `restore` to restore the files in **/home/bob** from a tape mounted in **/dev/rmt/t0**, with a blocking factor of 512, to a subdirectory under the current directory is

```
restore xvbfy 512 /dev/rmt/t0 ./home/bob
```

23.6.3.4 `dd`

The `dd` facility is used to copy a single file, part of a file, a partition, or part of a partition and can treat the data stream using, for example, compression or format conversion.

The basic syntax of `dd` is

```
dd if=device of=device bs=blocksize
```

where `if` is the input file pathname, `of` is the output file pathname, and `bs` specifies the block size.

An example of using `dd` in conjunction with `ssh` and `tar` is as follows.

```
$ ssh bob@192.168.0.13:/home/bob "dd if=backup.tar ibs=512" | tar
-xvBf
```

This command extracts the remote file **backup.tar** file at **/home/bob** in 512 byte blocks and streams it through `dd` to the system you typed this command on.

23.6.3.5 `ufsdump` *and* `ufsrestore` *(Solaris)*

`ufsdump` is an incremental file system dump facility for Solaris, similar to the `dump` command in PC-BSD. Its purpose is to back up all files specified (usually either a whole file system or files within a file system changed after a certain date) to magnetic tape or a disk file. The `ufsdump` command can only be used on unmounted file systems, or those mounted as read only. Attempting to dump a mounted, read–write file system might result in a system disruption or the inability to restore files from the dump. It is used in conjunction with `ufsrestore`.

Basic Syntax: `ufsdump ['options'] ['option arguments'] 'files _ to _ dump'`

Output: Backed-up files at a specified location

Examples:

Example 1: Using `ufsdump`

The following command makes a full dump of a root file system on **c0t3d0**, on a 150 MB cartridge tape unit 0:

```
# ufsdump 0cfu /dev/rmt/0 /dev/rdsk/c0t3d0s0
```

Example 2: Making an incremental dump

The following command makes and verifies an incremental dump at level 5 of the **usr** partition of **c0t3d0**, on a 1/2 inch reel tape unit 1:

```
# ufsdump 5fuv /dev/rmt/1 /dev/rdsk/c0t3d0s6
```

Basic Syntax: `ufsrestore i | r | R | t | x [abcdfhlmostvyLT] ['archive _ file'] ['factor'] ['dumpfile'] ['n'] ['label'] ['time-out'] ['filename']...`

where `filename` specifies the pathname of files (or directories) to be restored to disk.

`Ufsrestore` is an incremental file system restore facility used in conjunction with the `ufsdump` facility.

23.6.3.6 Script Files for Backup and Recovery

As shown in Chapter 16, user-written script files can be deployed by the system administrator to quickly and efficiently do backup and archiving. Whether they are coded in Python as shown in Chapter 16, or in a shell programming language that embeds any of the preceding command line facilities, they can be used to facilitate and automate the backup, recovery, and archiving of files, directories, or file systems. Additionally, there are several online sources for backup and archiving script files available, and we will not justify the need for planning, coding, or maintaining such script files versus using an online, ready-made program.

The following are two examples of script files, the first one a legacy application of `tar`, and the next one a modern UNIX program for doing backup and archiving. One is coded as Bourne shell scripts, and one is coded as a Python script. They work equally well on PC-BSD or Solaris, with the minor differences in coding for those two systems duly noted in the code.

Example 23.6: Simple Bourne Shell Example for Automating Tar Backups

The following Bourne shell script will backup **/home/bob** to a directory previously created named **simple_backup** in a compressed format. Use an editor of your choice to create it, and name the script file **backup.sh**:

```
#!/bin/sh
# To backup additional directories, put more pathnames in the
following command,
# separated by spaces, such as /home/bob /home/sarwar /home/
blahblah.
backup_source="/home/bob"
backup_destination="/home/bob/simple_backup"
filename="back1.tgz"
echo "Backing up your UNIX home directory"
tar cvpzf $backup_destination/$filename $backup_source
echo "Backup Complete"
```

EXERCISE 23.16

Give the commands necessary to automate the preceding script file using cron, to run the **backup.sh** script at 3:00 a.m. every week. Feel free to modify the time, day, and source and destination directories and files, so that you can use the script file to backup information important to you on your UNIX system.

Example 23.7: Extended Python Script Example Using `rsync` to do a "Rolling" Backup

This example is similar to Chapter 16, Example 16.30, with the notable exceptions that it is more extensive and copies more hard-linked files and directories across a network using ssh.

Basically, there are five operations the Python script file performs:

1. Checks to see if several numbered directories exist. If they don't exist, it creates them.
2. Removes the last or oldest directory.
3. Hard copies directory **1** into directory **2**. A hard copy makes a copy in which the two files share the same disk space (i.e., your files take up no extra room).
4. Uses `rsync` to backup the files you want to directory **1**. `rsync` only overwrites changed files. This example goes through local directories and then networked files.
5. Backs up the source code.

You should create a file in **/home/me/.rsync/exclude** with patterns for `rsync` to ignore (see the `rsync` manual for more info). We mainly exclude patterns like ***.tmp** or ***.o**. Also, you should setup ssh so that you can do a login to your server without having to type in a password. Otherwise this won't work as a cron task.

```
#!/usr/bin/python
#
import time
import datetime
import os
import shutil
import UserString
def backupserver():
    debug_flag = "debug"
    sources = ["/etc/aliases", "/etc/resolv.conf", "/etc/hosts",
              "/etc/named.conf", "/etc/group","/etc/dovecot.conf",
              "/var/www/cgi-bin","/etc/httpd","/var/www/html"]
    localsources = ["/Volumes/NO NAME/onix", "/Volumes/NO NAME/
                   projects", "/Users/me/myprojects",
                   "/Users/me/bin"]
```

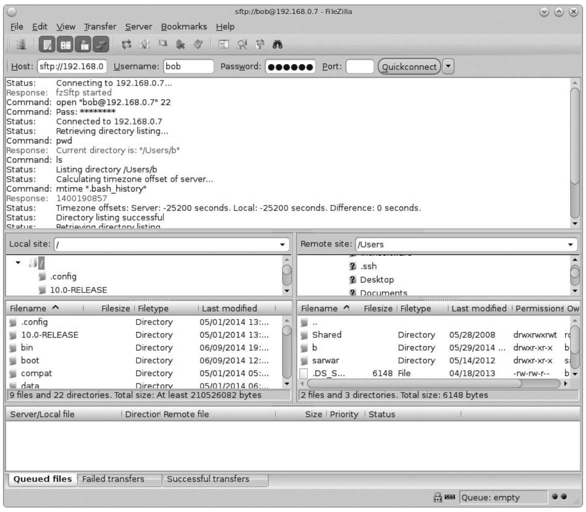
```
localcode = ["/Volumes/NO NAME/projects" ]
codetarget = "/Volumes/PC Backups/Code Backups/"
target = "/Volumes/PC Backups/Daily Backups/"
host = "www.mysite.com"
user = "me"
# target can be reached, trigger a rotation of the snapshot
# directories
if (debug_flag == "debug"):
    print "Date: " + str(datetime.date.today())
    print host +" is up, rotating snapshots."
# check to see if directories exist
i = 1
while i <= 5:
    temp_path = target + str(i) + "/"
    if not os.path.exists( temp_path ):
        try:
            os.makedirs( temp_path )
            print "Created " + temp_path
        except:
            print "Couldn't create " + temp_path
    i = i + 1
# cycle backups
# first delete #5
print "Deleting oldest archive"
shutil.rmtree( target + "5" )
# cycle 2 - 4
print "Cycle backups"
os.rename( target + "4", target + "5")
os.rename( target + "3", target + "4")
os.rename( target + "2", target + "3")
# do hard copy of 1
os.system('find "' + target + '1" -print | cpio -pdl ' +
target +"2")
print "Copy first backup"
# copy tree does a full copy whereas cpio does hard-link
# copies (i.e., each copied file
# takes up no extra space)
shutil.copytree( target + "1", target + "2")
os.system('cd "' + target + '1"; find . -print | cpio -pdl "'
+ target + '2"')
print "Rsyncing"
# Rsync from local directories to local backup
for source in localsources:
    print "Local directories " + source
    os.system('rsync -azv -e --delete --delete-excluded ' +
       '--exclude-from=/Users/me/.rsync/exclude "' + source +
        '" "' + target+'1"')
    # Rsync from the server to the local backup
    for source in sources:
       print "Downloading " + source
       os.system('rsync -azv -e ssh --delete --delete-excluded ' +
            '--exclude-from=/Users/me/.rsync/exclude ' + user +
            "@" +                host + ':"' + source + '" "' +
            target+'1"')
    # Backup *just* the programming code
    newfolder = codetarget + str(datetime.date.today())
    # make the new directory
    if not os.path.exists( newfolder ):
        os.makedirs( newfolder )
    # Here's the key line. Find all the source files from our
    # rsync backup and copy them as
    # hard links.
    print "Backing up source"
    os.system('cd "' + target + '1"; find . \( ' + " -name '*.cpp'
    -or -name 'Makefile'" +
        "-or -name '*.c' -or -name '*.h' -or -name '*.lex' -or
        -name '*.y'" +
        " -or -name '*.bat' -or -name '*.py'\) " + ' -print | cpio
        -pdl "' + newfolder +'"')
backupserver()
```

23.6.3.7 Software for Backup and Recovery: The "Zillas" and Ghost

There are many commercial software and hardware packages that can be deployed to backup and archive your system. One very easy-to-use commercial product that allows you to clone entire hard disks in a "broadcast" fashion over a network, from a server to one or more machines, is Norton Ghost. But the two most readily available, useful, and free software facilities that can do a variety of file system backup and recovery, disk recovery, and disk cloning operations are FileZilla and Clonezilla.

23.6.3.7.1 FileZilla FileZilla is nominally a graphics-based ftp client and server program that can use ssh as the tunnel, or conduit between systems. It has a number of useful functions and menu choices that allow the system administrator to successfully, confidentially, and efficiently backup and restore single files or directories, via a network globally. It is most useful for backing up and restoring single-user files and directories. It is not a replacement or substitute for the command line facilities shown in the preceding sections. Figure 23.17 illustrates the screen display and menus available in the PC-BSD "client" version of FileZilla, which can be installed quickly and easily via the AppCafe.

Both client and server, in our case a local machine running PC-BSD, and a remote UNIX machine, must have ssh communications protocol enabled between them. You can have login access to an account on the remote server, or you can anonymously login as well if that is enabled.



```
Status:     Connecting to 192.168.0.7...
Response:   fzSftp started
Command:    open "bob@192.168.0.7" 22
Command:    Pass: ********
Status:     Connected to 192.168.0.7
Status:     Retrieving directory listing...
Command:    pwd
Response:   Current directory is: "/Users/b"
Command:    ls
Status:     Listing directory /Users/b
Status:     Calculating timezone offset of server...
Command:    mtime ".bash_history"
Response:   1400190857
Status:     Timezone offsets: Server: -25200 seconds. Local: -25200 seconds. Difference: 0 seconds.
Status:     Directory listing successful
Status:     Retrieving directory listing
```

```
Local site: /
  /
    .config
    10.0-RELEASE

Filename ^      Filesize  Filetype    Last modified
.config                   Directory   05/01/2014 13:...
10.0-RELEASE              Directory   05/01/2014 13:...
bin                       Directory   06/09/2014 19:...
boot                      Directory   06/09/2014 12:...
compat                    Directory   05/01/2014 05:...
data                      Directory   05/01/2014 06:...
9 files and 22 directories. Total size: At least 210526082 bytes
```

```
Remote site: /Users
    .ssh
    Desktop
    Documents

Filename ^   Filesize  Filetype    Last modified    Permissions  Ow
..
Shared                 Directory   05/28/2008       drwxrwxrwt   ro
b                      Directory   05/29/2014 ...   drwxr-xr-x   b
sarwar                 Directory   05/14/2012       drwxr-xr-x   s
.DS_S...     6148      File        04/18/2013       -rw-rw-r--   b
2 files and 3 directories. Total size: 6148 bytes
```

```
Server/Local file    Directio Remote file    Size  Priority  Status
```

FIGURE 23.17 FileZilla main window.

After launching FileZilla on the client, to login to a remote host server you need to supply the IP address of the server, the login name and password, and the port number (22 for ssh).

Once you have successfully logged in, the local machine's directory and file structure is shown on the left side of the figure. The remote machine's directory structure is shown on the right side of the figure. To transfer files or directories between machines, you simply drag and drop between the appropriate panes on the left or right. If you are overwriting previously transferred files or directories, the FileZilla default is to give you the chance to overwrite or rename the files being transferred.

There are a number of other menu choices at the top of the FileZilla screen that allow you to affect preferences, set bookmarks, etc. For example, via the menu choice **Manage Bookmarks** and the Site Manager, you can automatically make multiple local directories and remote directories available for ssh transfer as soon as you log in to the remote server sites.

23.6.3.7.2 Clonezilla Clonezilla is the most useful and readily available tool for the ordinary single computer user to do partition and whole-disk cloning. The "live" version of it can be deployed to do non-broadcast cloning of entire disks, including the boot sector. This technique allows an ordinary user or a system administrator to take fast snapshots of entire disks, thus greatly enhancing and facilitating any backup strategy employed.

As an example, on a large single disk, Clonezilla is capable of quickly duplicating the entire disk so that the clone can be used in exactly the same way as the original disk. So, instead of using any of the other backup schemes shown above, at any single instant in time, you can create an exact copy of any of the disks on your system. Of course, you would have to weigh the merits of this versus any of the other backup strategies.

For PC-BSD, cloning the entire system disk, including the boot partition, is possible in Clonezilla. This would allow you to have a bootable duplicate of your system disk, possibly with all of your user data files on it as well. In CloneZilla, the source disk can be an internal hard drive, and the target clone can be an externally mounted hard disk in a USB-connected enclosure. This is a very useful procedure if you are running PC-BSD on a laptop computer that is only capable of having one internal hard drive.

In Chapter 24, Example 24.4, "Mirroring of Hard Disks on PC-BSD," we show a ZFS method (using mirroring) of doing exactly the same thing that CloneZilla does. Example 24.4 assumes the system disk and the disk you want to clone onto are internally mounted on the SATA bus. We also present a problem at the end of Chapter 24 (Problem 24.13) that asks you to repeat the procedures of Example 24.4 using the internally mounted system disk as the source and an externally mounted USB disk as the target for ZFS mirroring.

For Solaris, cloning partitions other than the boot partition is possible, but at this time, bootable system disk cloning is not possible.

23.7 SYSTEM UPGRADES AND SOFTWARE UPDATES USING A PACKAGE MANAGER

After a UNIX system has been installed and configured, two critical ongoing tasks for the system administrator are upgrading the system to incorporate *patches*, or fixes, that keep the system itself working optimally, and adding and updating the application software that users rely on to do their work. This section will show how to upgrade the parts of or the entire operating system and add/update the application software via package management. This will be done in the subsections that follow:

23.7.1: Upgrading the Operating System in Solaris 11

23.7.2: Updating the Installed Application Packages and Installing New Application Packages in Solaris

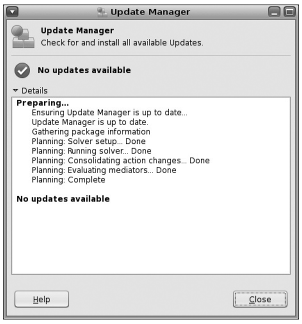
23.7.3: Upgrading the Operating System in PC-BSD

23.7.4: Updating the Installed Application Packages and Installing New Application Packages in PC-BSD.

Simply stated, a *package* is ready-to-install software that, as will be shown, installs itself with a minimum of administrator or user interference. Both PC-BSD and Solaris provide several ways of upgrading the operating system itself, and installing and updating application software. It is also possible to obtain the source code for system upgrades or application software updates, and via the program development process, compile, link, and assemble the source modules without using a package manager. This technique is not shown in this section.

23.7.1 Upgrading the Operating System in Solaris

The easiest and quickest way to achieve an upgrade of the operating system is to use the Update Manager, which is launched by making the pull-down menu choice



```
Update Manager
Check for and install all available Updates.

No updates available

Details
Preparing...
    Ensuring Update Manager is up to date...
    Update Manager is up to date.
    Gathering package information
    Planning: Solver setup... Done
    Planning: Running solver... Done
    Planning: Consolidating action changes... Done
    Planning: Evaluating mediators... Done
    Planning: Complete

No updates available
```

FIGURE 23.18 Update Manager window.

System>Administration>Update Manager. The Update Manager GUI appears on screen, as shown in Figure 23.18.

As seen in Figure 23.18, no updates are currently available for the system. If upgrades were available, a dialog would appear in the main pane of the System Manager window showing you the availability of new system upgrades. You would then be allowed to accept or decline downloading and installation of the available software modules. Then, if you accept, their download status and installation status would be shown. They would automatically be installed, and you would have to reboot the system before they took effect.

The `pkg` command allows you to add, remove, and update existing software packages on the system, including upgrading the operating system itself. Section 23.7.2 will give a brief introduction to the `pkg` command and its options that allow you to update the user application package software.

You can type the following two lines as superuser on the command line in a terminal window to see the version numbers of available operating system upgrades, and the upgrade release you are currently working on:

```
root@solaris:~# pkg list -af entire
NAME (PUBLISHER)                    VERSION                  IFO
entire                 0.5.11-0.175.1.0.0.24.2                i--
entire                 0.5.11-0.175.0.10.1.0.0                ---
entire                 0.5.11-0.175.0.0.0.2.0                 ---
entire                 0.5.11-0.151.0.1                       ---
root@solaris:~# uname -v
11.1
```

The output of the first command shows that the latest upgrade release is 0.5.11-0.175.1.0.0.24.2, and the `i` in the IFO column means that you are on that release currently. The output of the second command shows that you are using major release 11.1 of Solaris.

Then, if a new major release is available to you without an Solaris service contract, you can type the following command to update to that new major release:

```
root@solaris:~# pkg update
```

If any major update is available at the default repository, the operating system will be updated automatically.

23.7.2 Updating the Installed Application Packages and Installing New Application Packages in Solaris

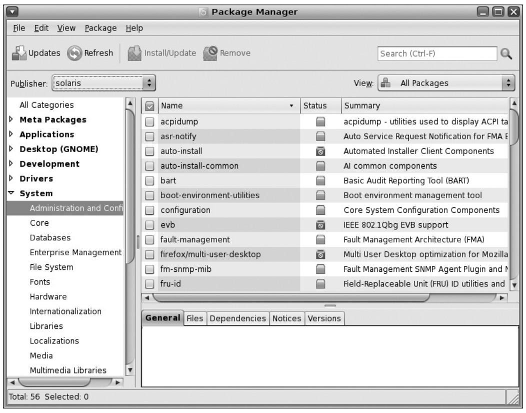
The easiest and most effective way to update or add new application packages to the system is with the Package Manager, which is launched by making the pull-down menu choice **System>Administration>Package Manager**. The Package Manager GUI appears on screen, as shown in Figure 23.19.

As seen in Figure 23.19, the repository (or publisher) that is being used is Solaris, the major package category selected is **System**, and the subcategory selected is **Administration and Configuration**. A total of 56 packages are in this category, none of which has been selected. To add a new package, choose one of the major categories, and then a subcategory, and examine the list of available packages. Under the **Status** column, you can see if a package has been installed already or has not.

In Figure 23.20, we show the choices you would make in order to install **tkinter-27**, the tkinter bindings to **tcl/tk** that were used in Chapter 16 to add GUI elements to Python programs. This package is found in the Solaris repository, in major package category **Development**, subcategory **Python**.

Before installing this package, you would have to install the **python-27** package shown in the listing above **tkinter-27** in the **Development** subcategory. Python 2.6 comes preinstalled with Solaris, so installing the **python-27** package first to overwrite Python 2.6 is necessary. As seen in Chapter 16, the main Python program, via importing, uses the **tkinter** library modules in a subsidiary way.

Highlight and put a check in the box next to the package and then click on the `Install/Update` button at the top of the screen. That package will be installed. You will see the progress of downloading and installation in the main pane of the `Package Manager` window.

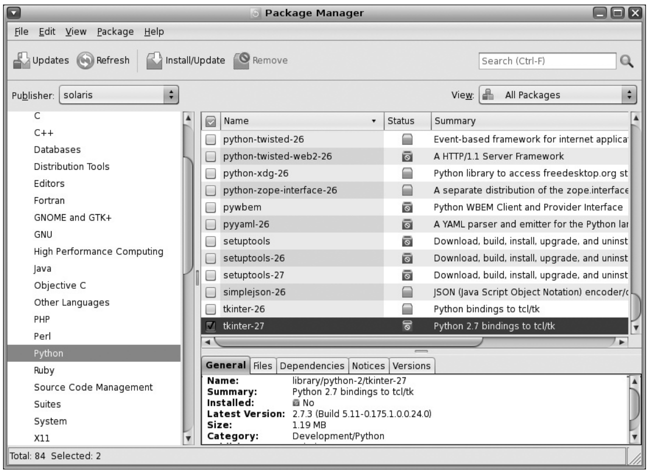


Publisher: solaris View: All Packages

Name	Status	Summary
acpidump		acpidump - utilities used to display ACPI ta
asr-notify		Auto Service Request Notification for FMA E
auto-install		Automated Installer Client Components
auto-install-common		AI common components
bart		Basic Audit Reporting Tool (BART)
boot-environment-utilities		Boot environment management tool
configuration		Core System Configuration Components
evb		IEEE 802.1Qbg EVB support
fault-management		Fault Management Architecture (FMA)
firefox/multi-user-desktop		Multi User Desktop optimization for Mozilla
fm-snmp-mib		Fault Management SNMP Agent Plugin and M
fru-id		Field-Replaceable Unit (FRU) ID utilities and

Total: 56 Selected: 0

FIGURE 23.19 Package Manager window.



Publisher: solaris

View: All Packages

C
C++
Databases
Distribution Tools
Editors
Fortran
GNOME and GTK+
GNU
High Performance Computing
Java
Objective C
Other Languages
PHP
Perl
Python
Ruby
Source Code Management
Suites
System
X11

☐	Name	Status	Summary
☐	python-twisted-26	▭	Event-based framework for internet applica
☐	python-twisted-web2-26	⊘	A HTTP/1.1 Server Framework
☐	python-xdg-26	▭	Python library to access freedesktop.org st
☐	python-zope-interface-26	▭	A separate distribution of the zope.interface
☐	pywbem	⊘	Python WBEM Client and Provider Interface
☐	pyyaml-26	⊘	A YAML parser and emitter for the Python lar
☐	setuptools	⊘	Download, build, install, upgrade, and uninst
☐	setuptools-26	⊘	Download, build, install, upgrade, and uninst
☐	setuptools-27	⊘	Download, build, install, upgrade, and uninst
☐	simplejson-26	▭	JSON (Java Script Object Notation) encoder/c
☐	tkinter-26	▭	Python bindings to tcl/tk
☑	tkinter-27	⊘	Python 2.7 bindings to tcl/tk

Name:	library/python-2/tkinter-27
Summary:	Python 2.7 bindings to tcl/tk
Installed:	⊘ No
Latest Version:	2.7.3 (Build 5.11-0.175.1.0.0.24.0)
Size:	1.19 MB
Category:	Development/Python

Total: 84 Selected: 2

FIGURE 23.20 Python **27-tkinter** package view in package manager window.

In a very similar manner, using a text-based command line interface, you can add, update, or remove packages that are already on your system using the pkg command and its options. Most importantly, you can more completely specify and control the addition, removal, and updating of user application software packages using pkg. In the following session, since we have previously determined that the name of the gcc compiler package for Solaris is gcc-48, we use the pkg command and install subcommand to add the gcc compiler to our system.

```
bob@solaris_11_3:~$ su
Password: xxx
root@solaris_11_3:~# pkg install gcc-48
           Packages to install: 17
           Mediators to change: 1
            Services to change: 1
       Create boot environment: No
Create backup boot environment: No
DOWNLOAD                  PKGS       FILES        XFER (MB)    SPEED
Completed                 17/17      2941/2941    66.2/66.2    337k/s
PHASE                                               ITEMS
Installing new actions                          3498/3498
Updating package state database                      Done
Updating package cache                                0/0
Updating image state                                 Done
Creating fast lookup database                        Done
Updating package cache                                1/1
root@solaris_11_3:~#
```

We can now use the command gcc to compile C programs.

23.7.3 Upgrading the Operating System in PC-BSD

The most reliable, fastest, time-efficient, and strategic way of upgrading PC-BSD at the current time is to follow the steps shown below in Section 23.7.3.1. Using the Update Manager, the traditional way of upgrading the PC-BSD system, is discouraged. This may change with time in later releases of the software, and we will make you aware of these changes on the authors' GitHub site for this book.

23.7.3.1 The Best Procedures for Upgrading PC-BSD

Contrary to the patented and officially recommended methods for upgrading your version of the PC-BSD operating system software, we advocate using the procedures of this section to accomplish a reliable upgrade. You may also wish to explore using the Update Manager or the `pkg` command and its various forms to do an incremental upgrade of the system software, or the FreeBSD base system that PC-BSD is built upon. But as a preliminary step before doing any of the procedures shown here, we recommend that you first set auto "Updates" (a misnamed operation according to standard usage of the term) to "None" in the Update Manager, and then quit the Update Manager.

The steps shown here are premised upon certain conditions that your computer may or may not fulfill. The procedures assume that you have at least two hard drives in your machine or the equivalent of that. This could mean that in a laptop, for example, you have some other high-speed medium that could serve as a second hard drive, such as an external USB 3.0 drive. Many available laptops have the capability to have two hard drives in them. Most desktop computers have the capability to add additional hard drives to them—either SSDs or the traditional spinning-disk variety—on the SATA bus. The steps shown apply most commonly to desktop or server-class machines. It is also possible in a contemporary computer to have SSD capability added on the PCI Express bus of a desktop system.

It is worth adding that the steps shown here can be applied to PC-BSD upgrades on systems installed in a virtual machine container, such as in VirtualBox or Docker. We do not detail the upgrade steps for those kind of installations, but we leave it to you to experiment in those kinds of environments to achieve the same results as shown here.

Certainly, the success of any operation shown in this section is based on the fact that you can actually install PC-BSD using an ISO-created DVD you have successfully obtained from www.pcbsd.org on your computer hardware. These procedures also work if you can install PC-BSD from a USB thumbdrive.

To gain a better understanding of how to use the `zpool` and `zfs` commands shown in the upgrade process steps, it is very important to first work through everything shown in Chapter 24.

The following upgrade process steps were carried out using an SSD as the original system disk and a spinning hard disk drive as the data disk:

1. Archive all of your user data on a removable medium, such as a USB thumbdrive.
2. Install the new release of PC-BSD from an ISO DVD downloaded from www.pcbsd.org. This effectively overwrites and destroys everything on your system disk. In our

case, the system disk was an SSD. This is probably the best choice for the main system disk and any other auxiliary disks, such as a separate disk to contain the ZFS Intent Log (ZIL).

3. Install a second hard drive into the computer. This drive must now be formatted by the PC-BSD Disk Manager, and a new zpool must be created on it using either ZFS commands on the command line or using the PC-BSD Disk Manager. Be aware that partitioning and formatting this new drive will effectively erase any data on that drive! These methods are covered completely in Chapter 24.
4. Once the second hard drive is partitioned and formatted, create ZFS datasets on the second hard drive to accommodate your archived user data.
5. Move the archived user data from step 1 onto the second hard drive, into the data sets you created in step 4.

The use cases where the above steps for upgrading PC-BSD might not work efficiently, as far as we can see, are if your original user data is (1) stored in a complex ZFS RAIDZ configuration on several disks, (2) found on volumes on multiple disks, (3) spread across local disks and cloud storage, or (4) inseparable from the operating system program in some way. But it is still possible to deal with those use cases in a hybridized version of the above five step procedure. If you want a reliable upgrade to the system software without jeopardizing your user data in any way, it is up to you to find the hybridized procedures that work best for your use case, and most importantly, your data storage model.

If you are installing PC-BSD for the first time, you might consider separating your operating system and your user data onto two physically different hard drives. That design would then allow you to operate according to the following hybridized upgrade process model:

1. On your SSD or spinning hard disk, install the PC-BSD system only. In our case, this was a smaller-capacity SSD, 120 GB in size. Be sure to size this system disk adequately to accommodate the PC-BSD installation of your choice.
2. Install another hard disk in your computer, if possible, and partition and format this second hard disk using the command line ZFS commands, or the PC-BSD Disk Manager appropriately. This hard disk will serve as your user data disk. You could use a single partition, formatted to BSD ZFS, or install multiple disks in a RAIDZ configuration at this point, partitioned and formatted similarly.
3. Create zpools on the user data disk or disks.
4. Create ZFS datasets on your disk or disks in the zpools from step 3.
5. Create your data in those ZFS datasets.
6. Use your system.

Now, to deploy this hybridized process when upgrading the PC-BSD system, do the following:

1. Export all of your data disk zpools using the `zpool export` command. This effectively allows you to transfer all of the zpools you have created for your data to a new system. In our case, that new system would be an upgraded version of PC-BSD resident on the same computer hardware.
2. Gracefully shutdown your system.
3. Install a new version of PC-BSD on the system disk from an ISO-created DVD obtained from www.pcbsd.org.
4. When the new version is installed, use the `zpool import` command to import all of the zpools you created on your data disk(s) in step 1.

Note very carefully here that we have found this procedure to work most reliably and most time efficiently as well. What you achieve using these techniques—which are much simpler than any of the patented and recommended methods, which, from our experience, are fraught with anecdotal and insurmountable problems for the beginner—is reliability and ease of use. Compare the procedures we show here to the various ways of using repositories for not only upgrading the operating system but updating the installed package base that you may have. If you find the patented and recommended ways easier to understand and use, then by all means use those.

23.7.3.2 Command Line Method of Manual Updates

If you wish to use the Update Manager to upgrade the operating system, for example to do FreeBSD base system upgrades to your PC-BSD system, we provide the following command line technique, which uses the `pc-updatemanager` command. Many traditional UNIX users like to use a command line utility to do upgrades. If you type the following on the command line, it will show its available options:

```
%pc-updatemanager
pc-updatemanager Usage
-----------
branches                - List available system branches
chbranch <tag>          - Change to new system branch
check                   - Check for system updates
install <tag>,<tag2>    - Install system updates
pkgcheck                - Check for updates to packages
pkgupdate               - Install packages updates
syncconf                - Update PC-BSD pkgng configuration
confcheck               - Check PC-BSD pkgng configuration
-j <jail>               - Operate on the jail specified
```

The PC-BSD Update Manager will automatically upgrade the operating system and upgrade installed packages, if you have configured it in any of the possible ways except the

"None" option. The "None" option and removing the Update Manager icon from the System Tray by quitting is, from the authors perspective, the most effective way of doing upgrades. It is possible to do upgrades of the operating system and related software packages by using the sequence of commands shown in the following command line session, but even that is not advisable at this time. The surest, most reliable, and time-efficient way to do operating system upgrades is to use the techniques shown in Section 23.7.3.1 This is particularly true when doing a major update—for example, between release 10 and release 11 of the PC-BSD system.

If you still want to update the FreeBSD base system on the command line, for example, we provide the following short command line session, which allows you to do an upgrade of the FreeBSD base system, using the `pc-updatemanager` command.

```
% sudo pc-updatemanager check
```

The system informs you that an upgrade for FreeBSD is available. This particular form of the command was done to upgrade the FreeBSD base system from 10.1 to 10.2:

```
%sudo pc-updatemanger install fbsd-10.2-RELEASE
```

After FreeBSD downloads and installs automatically, it is necessary to reboot the system.

```
%sudo reboot
```

After rebooting the computer, if you have the Update Manager set to "Automatic," and "All," packages and security updates for the PC-BSD system will be done automatically.

If no system updates are available, the `check` command will indicate: `Your system is up to date!`.

To determine if package updates for the PC-BSD system are available, use this command:

```
% sudo pc-updatemanager pkgcheck
Updating repository catalogue
Upgrades have been requested for the following 253 packages:
... [Output truncated] ...
The upgrade will require 70 MB more space
439 MB to be downloaded
To start the upgrade run "/usr/local/bin/pcupdatemanager
pkgupdate"
```

In the above example output, newer versions are available for 253 packages. The list of package names was truncated in the sample output. If no updates were available, the output would have instead said `All packages are up to date!`.

If updates are available, you can install them with this command:

```
% sudo pc-updatemanager pkgupdate
Updating repository catalogue
```

```
snip downloading and reinstalling output
[253/253] Upgrading pcbsdbase
from 1374071964 to 1378408836... done
Extracting desktop overlay data... DONE
```

The output has been truncated in this example. The package update process will download the latest versions of the packages which need updating, displaying the download progress for each file. Once the downloads are complete, it will display the reinstallation process for each file. The last step of the package update process is to extract the desktop (or server) overlay and then to return the prompt. After performing any updates, reboot the system.

23.7.4 Updating the Installed Application Packages and Installing New Application Packages in PC-BSD

AppCafe is a GUI tool for installing and removing application software, and as such allows you to quickly and easily add a wide range of programs to the installed PC-BSD base in one easy step. You launch AppCafe from the PC-BSD desktop by clicking once on its icon. You can also use AppCafe to easily update or remove any installed application software package on your system.

The GUI window of AppCafe is very intuitive, and allows you to see what software packages are available and ones that are already installed. Using the `Categories` pane of the GUI window, you can choose from general areas within which a desired type of application software package might be found. If you click on any of the icons presented in the GUI window, you get more information about the software, and are also able to install it. You can search for an application software package from among those available at the AppCafe repository by using the `App Search` icon and typing in the exact name of the application desired.

The installation of updates and new software packages with AppCafe is highly version dependent. Therefore, the best procedure to follow to update or install new software packages is to refer to the PC-BSD handbook that comes with your release of the software to find the most current detailed steps for accomplishing these procedures. Methods of updating, installing, and removing packages are more completely detailed in the PC-BSD handbook.

There is also a way to manually update packages on your PC-BSD system using the `pkg` command. We do not show examples of this, but you can refer to both the man pages for the `pkg` command and the version of the PC-BSD handbook that comes with your system.

23.8 SYSTEM AND SOFTWARE PERFORMANCE MONITORING

The most important considerations the UNIX system manager has to make when dealing with system performance revolves around CPU process management, memory management, disk usage/management, and network performance. Table 23.8 lists the controlling facilities and functions UNIX provides for system tuning and performance monitoring.

TABLE 23.8 Performance-Tuning Functions

System Component	Control Facility
CPU	Nice numbers
	Process priorities
	Batch queues
	Scheduler parameters
Memory	Process resource limits
	Memory management parameters
	Paging space
ZFS vdevs, zpools and I/O	ZFS pool and file system organization
	ZFS deduplication, efficiency, optimization
	I/O parameters
Network I/O	Network memory buffers
	Network-related parameters
	Network infrastructure

The commands and configuration files that implement and affect some of the functions from Table 23.8 are

```
For PC-BSD- ps, sysctl, /etc/sysctl.conf
For Solaris- ps, dispadmin, ndd, /etc/system
```

23.8.1 Process and Memory Management

The most important, complete, and readily available display of system activity is given by the ps command. The following code example shows you the display of the nine top current processes using the most CPU resources running on the system. When we type this in on our PC-BSD system, we get the output shown:

```
[bob@pcbsd-4976] ~% ps aux | head -10
USER     PID  %CPU %MEM     VSZ     RSS    TT  STAT  STARTED     TIME COMMAND
root      11 198.0  0.0       0      32 -  RL        Fri05PM    5437: [idle]
                                                                07.08
root    6737   2.0  1.0  162592   61176 -  I<        Fri05PM  6:50.26 X :0 -auth
                                                                      /tmp/.
                                                                      PCDMA
bob    99659   1.8  0.1   23656    3948 1  Ss         2:51PM  0:00.15 /bin/csh
bob    99654   1.1  1.1  505084   66592 -   S         2:51PM  0:00.55 kdeinit4:
                                                                      kdeinit4:
                                                                      kk
bob    17229   0.8  2.6 1047556 157644 -    I        Fri05PM  1:01.80 kdeinit4:
                                                                      kdeinit4:
                                                                      pdt
root       0   0.0  0.1       0    4880 -  DLs       Fri05PM  5:52.47 [kernel]
root       1    0.0   0.0     9432     756  -  SLs       Fri05PM  0:00.08 /sbin/init--
root       2    0.0   0.0        0      16  -   DL       Fri05PM  0:00.00 [crypto]
root       3    0.0   0.0        0      16  -   DL       Fri05PM  0:00.00 [crypto
                                                                          returns]
[bob@pcbsd-4976] ~%
```

A similar output can be obtained in Solaris by giving the following command:

```
%ps -ef | head -10
```

The `pgrep` command displays the PIDs of running processes. Here are some examples of how to use `pgrep` to find the process IDs of the running processes and pipe those PIDs to another command to produce the output.

Search for **kdeinit** and run `ps` (assumes you are running the KDE4 desktop environment).

```
$ ps -p 'pgrep kdeinit'
PID        TT STAT      TIME          COMMAND
17189   -        Is   00:00.96        kdeinit4: kdeinit4 Running…
(kdeinit4)
... [Output truncated]
$
```

Search for **okular** and run `ps`

```
$ ps -fp 'pgrep okular'
PID       TT STAT       TIME          COMMAND
46917      -  I      0:27:14          okular /usr/local/share/pcbsd/
doc/handbook_en.pdf
$
```

Search for **okular**, improve its priority (assumes Okular, the PDF viewer, is running and you are the superuser).

```
$ renice -1 'pgrep okular'
46917: old priority 0, new priority -1
$
```

The `nice` and `renice` commands, as seen in the previous example, change process priorities in the CPU.

Here is an example of using a command with `nice` to change a command's *nice* value.

Launch FileZilla at higher priority:

```
$ nice -n -1 filezilla
$
```

When a process is already running, you can change the process's nice value using the `renice` command. Here are some examples of the `renice` command.

Renice **sarwar**'s processes +2:

```
$ renice +2 -u sarwar
$
Renice PID 2576 by +5
$ renice +5 2576
$
```

Renice **sarwar**'s ksmserver processes to –3:

```
$ renice -3 'pgrep -u sarwar ksmserver '
2545: old priority -1, new priority -3
2546: old priority -1, new priority -3
2547: old priority -1, new priority -3
```

The back quotes are used in the previous command lines to indicate that the output of the `pgrep` command (presumably PIDs) be fed to the `nice` and `renice` command. The nice settings for your processes are displayed by default when you run `top`.

In PC-BSD, you can control kernel parameters with the system running using the `sysctl` command. You can also add parameters permanently to the **/etc/sysctl.conf** file, so they can load as a group or at each reboot. Some useful examples are as follows.

List all kernel parameters:

```
$ sysctl -a | less
kernel.ostype: FreeBSD
kernel.osrelease: 10.0-RELEASE-p12
kern.osrevision: 199506
kern.version: FreeBSD 10.0-RELEASE-p12 #0: Wed June 4 14:50:48 UTC
2014
... [Output truncated]
```

List the value of particular parameter:

```
$ sysctl kern.hostname
kern.hostname:
```

For our system, it was listed as **pcbsd-4976**.

Set the value of `kernel.geom.debugflags` to 16:

```
$ sysctl kern.geom.debugflags=16
```

If you want to change any of your kernel parameters permanently, you should add them to the **/etc/sysctl.conf** file. Parameter settings in that file are in the form `parameter = value`.

For Solaris, the `dispadmin` command displays or changes process scheduler parameters while the system is running. Some useful examples using `dispadmin` are as follows.

Retrieves the current scheduler parameters for the real-time class from kernel memory and writes them to the standard output (time quantum values are in microseconds):

```
$ dispadmin -c RT -g -r 1000000
# Real Time Dispatcher Configuration
RES=1000000
# TIME     QUANTUM                  PRIORITY LEVEL
# (rt_quantum)
1000000                #                        0
1000000                #                        1
... [Output truncated]
```

Retrieves the current scheduler parameters for the time-sharing class from kernel memory and writes them to the standard output (time quantum values are in nanoseconds):

```
$ dispadmin -c TS -g -r 1000000000
# Time Sharing Dispatcher Information
RES=1000000000
#ts_quantum   ts_tqexp   ts_slpret   ts_maxwait
ts_lwait  PRIORITY  LEVEL
2000000000     0         50      0         50      #        0
... [Output truncated]
```

23.8.2 Disk Usage and Management

The following are samples of the ZFS commands from Chapter 24, and the traditional UNIX du command, that allow a system administrator to monitor the disk usage and status of the system disks.

- `zpool list`: Lists existing or all or indicated pools, along with health status and space usage
- `zpool status`: Displays the detailed health status of all or indicated pools
- `zpool iostat`: Displays I/O statistics for all or indicated pools
- `zpool upgrade -a`: Upgrades all pools to the latest available version
- `zpool set deduplication on/off`: Sets the deduplication property of a pool
- `zpool scrub`: Examines all data in the named pool and verifies that the checksums are correct

- `zfs upgrade`: Upgrades file systems to a new on-disk version
- `du`: Displays the block usage for file systems and directories

For ZFS, PC-BSD is currently at version 5000, Solaris 11 at version 34.

The following set of commands show the output of the preceding commands executed on a PC-BSD system. The output for a Solaris system is very similar, using the default pool name of **rpool** instead of **tank**.

```
[bob@pcbsd-4976] /usr/home/bob# zpool list
NAME      SIZE  ALLOC   FREE    CAP  DEDUP  HEALTH  ALTROOT
sender  95.5M   374K  95.1M     0%  1.00x  ONLINE  -
tank      928G  10.5G   917G     1%  1.00x  ONLINE  -
[bob@pcbsd-4976] /usr/home/bob# zpool list tank
NAME   SIZE  ALLOC   FREE    CAP  DEDUP  HEALTH  ALTROOT
tank   928G  10.5G   917G     1%  1.00x  ONLINE  -
[bob@pcbsd-4976] /usr/home/bob# zpool iostat
               capacity     operations    bandwidth
pool        alloc   free   read  write   read  write
----------  -----  -----  -----  -----  -----  -----
sender       374K  95.1M      0      0     15      2
tank        10.5G   917G      0     17  9.92K   120K
----------  -----  -----  -----  -----  -----  -----
[bob@pcbsd-4976] /usr/home/bob# zpool iostat tank
               capacity     operations    bandwidth
pool        alloc   free   read  write   read  write
----------  -----  -----  -----  -----  -----  -----
tank        10.5G   917G      0     17  9.91K   120K
[bob@pcbsd-4976] /usr/home/bob# zpool status
  pool: sender
state: ONLINE
  scan: none requested
config:
        NAME                    STATE     READ WRITE CKSUM
        sender                  ONLINE       0     0     0
          /usr/home/bob/master  ONLINE       0     0     0
errors: No known data errors
  pool: tank
state: ONLINE
  scan: resilvered 9.09G in 0h5m with 0 errors on Fri May 2
20:55:14 2014
config:
        NAME        STATE     READ WRITE CKSUM
        tank        ONLINE       0     0     0
          ada0s1a   ONLINE       0     0     0
```

```
errors: No known data errors
[bob@pcbsd-4976] /usr/home/bob# zfs list
NAME                        USED  AVAIL  REFER  MOUNTPOINT
sender                      324K  63.2M    33K  /sender
sender/bacup                70K  63.2M  31.5K  /sender/bacup
sender/data                 52K  63.2M  31.5K  /sender/data
tank                        10.5G   903G   144K  legacy
tank/ROOT                   9.18G   903G   144K  legacy
tank/ROOT/default           9.18G   903G  9.18G  /mnt
tank/tmp                    396K   903G   396K  /tmp
tank/usr                    1.34G   903G   144K  /mnt/usr
tank/usr/home               170M   903G   160K  /usr/home
tank/usr/home/bob           168M   903G   168M  /usr/home/bob
tank/usr/home/sarwar        1.84M   903G  1.84M  /usr/home/sarwar
tank/usr/jails              144K   903G   144K  /usr/jails
tank/usr/obj                144K   903G   144K  /usr/obj
tank/usr/pbi                1.18G   903G  1.18G  /usr/pbi
tank/usr/ports              296K   903G   152K  /usr/ports
tank/usr/ports/distfiles   144K   903G   144K  /usr/ports/distfiles
tank/usr/src                144K   903G   144K  /usr/src
tank/var                    5.36M   903G   144K  /mnt/var
tank/var/audit              160K   903G   160K  /var/audit
tank/var/log                580K   903G   580K  /var/log
tank/var/tmp                4.50M   903G  4.50M  /var/tmp
[bob@pcbsd-4976] /usr/home/bob# zpool upgrade
This system supports ZFS pool feature flags.
All pools are formatted using feature flags.
Every feature flags pool has all supported features enabled.
[bob@pcbsd-4976] /usr/home/bob# zfs upgrade
This system is currently running ZFS filesystem version 5.
All filesystems are formatted with the current version.
[bob@pcbsd-4976] /usr/home/bob# zpool status sender
  pool: sender
state: ONLINE
  scan: scrub repaired 0 in 0h0m with 0 errors on Thu Jun 19
19:44:29 2014
config:
        NAME               STATE       READ    WRITE   CKSUM
        sender             ONLINE       0         0       0
          /usr/home/bob/master  ONLINE 0         0       0
errors: No known data errors
[bob@pcbsd-4976] /usr/home/bob# du > dufile
[bob@pcbsd-4976] /usr/home/bob# more dufile
18      ./.i3
54      ./.gnome/apps
62      ./.gnome
```

```
9       ./.sylpheed-2.0/newscache
9       ./.sylpheed-2.0/plugins
14      ./.sylpheed-2.0/uidl
9       ./.sylpheed-2.0/imapcache
9       ./.sylpheed-2.0/mimetmp
9       ./.sylpheed-2.0/tmp
132     ./.sylpheed-2.0
13      ./Images
5872    ./Downloads
13      ./.dbus/session-bus
22      ./.dbus
12432   ./Desktop/unix
12477   ./Desktop
13      ./Music
13      ./.gconf/desktop/gnome/background
22      ./.gconf/desktop/gnome
30      ./.gconf/desktop
39      ./.gconf
13      ./.putty
... [Output truncated]
```

23.8.3 Network Configuration

The most important and useful command for the system administrator in postinstall network configuration are the `ifconfig` command in PC-BSD and the `ipadm` command in Solaris. The following sections illustrate the basic usage of these commands.

23.8.3.1 `ifconfig` *in PC-BSD*

To view the syntax of the `ifconfig` command, type any invalid option on the command line as follows:

```
$ ifconfig -8
ifconfig: illegal option -- 8
usage: ifconfig [-L] [-C] [-g groupname] interface address_family
[address [dest_address]]
                [parameters]
       ifconfig interface create
       ifconfig -a [-L] [-C] [-g groupname] [-d] [-m] [-u] [-v]
[address_family]
       ifconfig -l [-d] [-u] [address_family]
       ifconfig [-L] [-C] [-g groupname] [-d] [-m] [-u] [-v]
$
```

To see a listing of the NICs attached to your system, type the following:

```
$ ifconfig -l
bge0 lo0
$
```

To see the configuration of the NIC **bge0**, type the following:

```
$ ifconfig bge0
bge0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0
mtu 1500
options=c019b<RXCSUM,TXCSUM,VLAN_MTU,VLAN_HWTAGGING,VLAN_HWCSUM,
TSO4,VLAN_HWTSO,LINKSTATE>
        ether e4:11:5b:12:c2:77
        inet6 fe80::e611:5bff:fe12:c277%bge0 prefixlen 64 scopeid
0x1
        inet 192.168.0.13 netmask 0xffffff00 broadcast
192.168.0.255
        nd6 options=23<PERFORMNUD,ACCEPT_RTADV,AUTO_LINKLOCAL>
        media: Ethernet autoselect (100baseTX <full-duplex>)
        status: active
```

To display a capabilities listing of NIC **bge0**, type the following:

```
$ ifconfig -m bge0
bge0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0
mtu 1500
        options=c019b<RXCSUM,TXCSUM,VLAN_MTU,VLAN_HWTAGGING,VLAN_
HWCSUM,TSO4,VLAN_HWTSO,LINKSTATE>
        capabilities=c019b<RXCSUM,TXCSUM,VLAN_MTU,VLAN_HWTAGGING,
VLAN_HWCSUM,TSO4,VLAN_HWTSO,LINKSTATE>
        ether e4:11:5b:12:c2:77
        inet6 fe80::e611:5bff:fe12:c277%bge0 prefixlen 64 scopeid
0x1
        inet 192.168.0.13 netmask 0xffffff00 broadcast
192.168.0.255
        nd6 options=23<PERFORMNUD,ACCEPT_RTADV,AUTO_LINKLOCAL>
        media: Ethernet autoselect (100baseTX <full-duplex>)
        status: active
        supported media:
                media autoselect mediaopt flowcontrol
                media autoselect
                media 1000baseT mediaopt full-duplex,master
                media 1000baseT mediaopt full-duplex
                media 1000baseT mediaopt master
                media 1000baseT
                media 100baseTX mediaopt full-duplex
                media 100baseTX
                media 10baseT/UTP mediaopt full-duplex
                media 10baseT/UTP
$
```

The ifconfig command can be used to change *NIC* settings as well as display them. A NIC is a Network Interface Card, what the system hardware uses to interface with a network. After executing each of the next series of ifconfig commands, view the changes in the NIC settings by using the ifconfig -l command.

To hard set the NIC **bge0** to 100 MBps with full duplex, type this:

```
$ ifconfig bge0 media 100baseTX mediaopt full-duplex
```

To hard set the speed to 10 MBps, type this:

```
$ ifconfig bge0 media 10baseT/UTP
```

To change the IP address and netmask, type this:

```
$ ifconfig bge0 inet 10.0.0.208 netmask 255.0.0.0
```

The changes just made to your NIC settings are good for the current session. When you reboot, however, those settings will be lost.

23.8.3.2 ipadm *in Solaris*

The ipadm command administers the network IP interfaces and TCP/IP parameters.

To display information about the current IP interface configuration, use the following option:

```
$ ipadm show-if
IFNAME     STATE    CURRENT      PERSISTENT
lo0        ok       -m-v------46 ---
bfe0       ok       bm--------46 ---
iwi0       ok       bm--------46 ---
$
```

To display the address associated with the IP interface, use the following option:

```
$ ipadm show-addr
ADDROBJ           TYPE     STATE       ADDR
lo0/v4            static   ok          127.0.0.1/8
bfe0/_b           dhcp     ok          ?
iwi0/_b           dhcp     ok          192.168.0.12/24
lo0/v6            static   ok          ::1/128
bfe0/_a           addrconf ok          fe80::212:3fff:fed1:8578/10
```

```
iwi0/_a          addrconf ok          fe80::212:f0ff:fe75:1442/10
$
```

23.8.4 Practical System Administration Logging and the `syslog`() Function

Logging and log files refer to the recording of general and specific actions and events on a UNIX system. PC-BSD and Solaris have very similar traditional methods available for logging. Logs are produced for a UNIX system administrator in order to audit the general operation of the system, particularly with regard to performance enhancement and the maintenance of system security.

A log is a record of the events occurring on a system. Logs are composed of log entries; each entry contains information related to a specific event that has occurred within the system. Originally, logs were used primarily for troubleshooting problems, but logs now serve many functions, such as optimizing system and network performance, recording the actions of users, and providing data useful for investigating malicious activity. Logs have evolved to contain information related to many different types of events occurring within networks and systems. Many logs contain records related to computer security; common examples of these computer security logs are audit logs that track user authentication attempts and security device logs that record possible attacks.

Because of the widespread deployment of networked servers, workstations, and other computing devices, and the ever-increasing number of threats against networks and systems, the number, volume, and variety of computer security logs has increased greatly. This has created the need for computer security log management, which is the process for generating, transmitting, storing, analyzing, and disposing of computer security log data.

Logs can contain a wide variety of information on the events occurring within systems and networks. Many logs created within the operating system environment could have some relevance to computer security. For example, logs from network devices such as switches and wireless access points, and from programs such as network monitoring software, might record data that could be of use in computer security or other IT initiatives, such as operations and audits, as well as in demonstrating compliance with regulations. However, for computer security these logs are generally used on an ad hoc basis as supplementary sources of information. The system administrator should consider the value of each potential source of computer security log data when designing and implementing a log management infrastructure.

Most of the sources of the log entries run continuously, so they generate entries on an ongoing basis. However, some sources run periodically, so they generate entries in batches, often at regular intervals.

The method of logging we show here is as follows.

- A system program, perhaps executed by a user, generates a call to write to a specific log file, either locally or across the network.
- The syslogd daemon handles that write call and is guided by entries in **/etc/syslog.conf**.

- A log entry is written in a specific location, usually to a log file in the **/var/log** directory. The log entry can be written to a log file on another computer system across the network as well.

For example, if someone requests superuser privilege on the system, syslogd writes that request to a log file. These are just a couple of simple cases, among many other logging processes that the system does for the administrator.

In this section we cover the traditional approach to doing general system logging, and briefly mention the more modern approach using `rsyslog`.

In detail, we cover:

- What the contents of the **/etc/syslog.conf** file look like and what a simple entry means
- Making changes to the **/etc/syslog.conf** file to customize it
- Restarting the syslogd daemon
- How a system program generates a log message
- The end of a log file

Controlling, and most importantly customizing, this general operation of logging for a particular installation is what a system administrator does. The process of custom modification of the system has two parts to it.

The first part is to make a modification to the configuration file named **/etc/syslog.conf**. This file designates what and where actions and events should be logged. Carefully specifying and controlling where log entries are logged is an important issue for system administrators. When a system administrator can quickly and efficiently examine critical log files it becomes much easier to audit the operations of the system. Log files for several machines and systems can exist on a single remote computer across the network to facilitate the audit process.

The second part is the rereading of this configuration file by the logging daemon syslogd, which actually executes the logging process. The syslogd daemon is always started when the system boots, and is always monitoring and logging events and actions specified in **/etc/syslog.conf**. Once you make a customizing modification to the default **/etc/syslog.conf**, you must then refresh (or stop and restart) syslogd in order for that change to take effect. We will show this for Solaris using an `svcadm` command, and for PC-BSD using the `service syslogd restart` command.

*23.8.4.1 The **/etc/syslog.conf** File*

Entries in **/etc/syslog.conf** have two fields, the *selector* field and the *action* field. The selector field is composed of one or more list elements separated by the semicolon character. Each list element has a facility and a level, separated by a period. The selector and action fields are separated by one or more tab characters, applicable to both PC-BSD and Solaris.

An example of a line with selector and action fields in it that you could place in **/etc/syslog.conf** on a Solaris or PC-BSD system is as follows.

```
user.notice  /var/log/syslog
```

In this example line, there is a single selector field facility, level-facility **user** program-generated messages, and level **notice**. The action field of the example is where the messages should be logged—in this case, **/var/log/syslog**.

In Solaris, we edited **/etc/syslog.conf** and placed the preceding example line in that file, and then refreshed the syslogd daemon by using the command:

```
%svcadm refresh system-log
```

After we made this change and refreshed syslogd, user programs that generated notice-level messages had those messages logged in **/var/log/syslog**.

In PC-BSD there was already a line in the default **/etc/syslog.conf** file that would log notice-level log entries by users to the file **/var/log/messages**. It would be instructive for you to find that line in the default PC-BSD **/etc/syslog.conf** file. By default, there was no file **/var/log/syslog** on our PC-BSD system, so we created an empty file with that name as superuser. Then, when we added the example line to **/etc/syslog.conf** on a PC-BSD system, and restarted syslogd, the example line in **/etc/syslog.conf** took effect.

The easiest way to stop and restart the syslogd daemon would be to use the command `service syslogd restart`. Restarting syslogd can also be done from the Service Manager in the PC-BSD Control Panel.

If you look in the default **/var/adm** or **/var/log** directories on a Solaris or PC-BSD system, there already exist sub directories and many different log files which allow for the compartmentalizing of logging messages, and their archiving over time. Instead of listing the contents of those directories, we encourage you to look at the contents of those directories and log files to gain an appreciation of what and where actions and events are currently being logged on your system. Then examine the contents of **/etc/syslog.conf** and try to make an association between log file contents and lines in the **/etc/syslog.conf** file.

Tables 23.9 and 23.10 list the generic facility and level descriptions allowed for both Solaris and PC-BSD.

The syslogd daemon takes messages from programs running either locally or remotely, and then writes the messages to a log file. The written message contains a time stamp, the kind of message it is, and the message itself.

23.8.4.2 How to Use the `syslog()` Function

The `syslog()` function, called perhaps from within a user-written system program, writes log messages via the syslogd daemon. The way to use this function is to open a log file, write to it with `syslog`, and then close the log file. The messages can then be written to the system console, log files, logged-in users, or forwarded to other machines as appropriate.

TABLE 23.9 Logging Levels

Name	Facility
`kern`	Kernel
`user`	Regular user processes
`mail`	Mail system
`lpr`	Line printer system
`auth`	Authorization system, or programs that ask for user names and passwords (`login, su, getty, ftpd`, etc.)
`daemon`	Other system daemons
`news`	News subsystem
`uucp`	UUCP subsystem
`local0... local7`	Reserved for site-specific use
`mark`	A timestamp facility that sends out a message every 20 minutes

TABLE 23.10 General Facilities

Priority	Meaning
`emerg`	Emergency condition, such as an imminent system crash, usually broadcast to all users.
`alert`	Condition that should be corrected immediately, such as a corrupted system database.
`crit`	Critical condition, such as a hardware error.
`err`	Ordinary error.
`warning`	Warning.
`notice`	Condition that is not an error, but possibly should be handled in a special way.
`info`	Informational message.
`debug`	Messages that are used when debugging programs.
`none`	Do not send messages from the indicated facility to the selected file. For example, specifying **.debug;mail.none* sends all messages except mail messages to the selected file.

The message is tagged with a priority. Priorities are encoded as a facility and a level, as previously described. The facility describes the part of the system generating the message. The level is selected from the following list (ordered high to low).

```
LOG_EMERG   A panic condition; normally broadcast to all users
LOG_ALERT   A condition that should be corrected immediately
(e.g., a corrupted system database)
LOG_CRIT    Critical conditions (e.g., hard device errors)
LOG_ERR     Errors
LOG_WARNING Warning messages
LOG_NOTICE  Conditions that are not error conditions, but should
possibly be handled specially
LOG_INFO    Informational messages
```

```
LOG_DEBUG   Messages that contain information normally of use only
when debugging a program
```

The `syslog()` function is used in conjunction with several other functions to open, write to, and then close a log file. The `openlog()` function provides for more specialized processing of the messages sent by `syslog()`. The `setlogmask()` function sets the log priority mask and returns the previous mask. The `closelog()` function can be used to close the log file. See the man page for `syslog` on your system for more details of these other functions.

Following is a simple example of a system program that writes two "notice" messages. The messages are written to the file designated in the example line addition to **/etc/syslog.conf** shown in Section 23.8.4.1:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <syslog.h>
int main(void) {
    int i;
    setlogmask (LOG_UPTO (LOG_NOTICE));
    openlog ("program_name", LOG_CONS | LOG_PID | LOG_NDELAY,
LOG_USER);
    syslog (LOG_NOTICE, "Program started by User %d", getuid ());
    syslog (LOG_NOTICE, "This is a simple message written to the
system log");
    closelog ();
    return 0;
```

If you run this program on PC-BSD, the two "notice" messages will be written to **/var/log/messages** by default. If you run this program on Solaris, the two "notice" messages will be written to **/var/log/syslog** by default.

23.8.4.3 Examining the End of a Log File

Since log files can become very large depending upon how much system activity there is, and how that activity is monitored, it is important for the system administrator to be able to quickly see the latest information in any particular log file.

The following code snippet displays only the last 25 lines of the **/var/log/messages** file. Change 25 to any number you want to have displayed:

```
% tail -25 /var/log/messages
```

To monitor the content of a particular log file in real time, the following command is useful. It lets you view a growing log file and see only the newer contents.

```
% tail -f /var/log/syslog
```

23.8.4.4 `rsyslog`

System logging can be accomplished in a more precise, capable, and extensive way using the `rsyslog` utility. By default, `rsyslog` is not installed on PC-BSD or Solaris systems. It can be installed by using the `pkg` management utility on both systems via the following command:

```
% pkg install rysylog
```

The syslogd daemon must then be disabled on both PC-BSD and Solaris before you can use `rsyslog`. To accomplish this in Solaris, use the following command:

```
% svcadm disable svc:/system/system-log:default
```

On PC-BSD, follow the instructions given at the end of the `pkg install` command. For more information on `rsyslog`, see the website www.rsyslog.com.

23.9 SYSTEM SECURITY

There are various techniques for securing a UNIX system, implemented from the single-user level on up to the most general system level. These techniques fall into the following areas:

- Password-Based Authentication
- Access Control: Discretionary (DAC), Mandatory (MAC), or Role-Based (RBAC)
- Using Access Control Lists (ACLs) in PC-BSD
- Intrusion Detection and Intrusion Detection Systems
- UNIX Security Software
- System Firewall

23.9.1 Password-Based Authentication

The first line of defense in system security, and the technique employed almost universally across many types of computer system, is password-based authentication. The UNIX system compares a user-entered password at login for a users ID, compares the password to a previously established and stored one held in a password file for that user, and based on the comparison authenticates the user. The ID not only determines whether the user can gain access to the system itself, but also determines what privileges the user has—for example, superuser privilege. Also, the ID is used in *discretionary access control* (DAC), as shown in the Section 23.9.2. The password file and a hash/salt scheme using a SHA256 hashing algorithm for encrypting the password, work together to authenticate a users ID. The password file on the system, in **/etc/passwd**, holds user information and works in conjunction with the **/etc/shadow** (on Solaris) or **/etc/master.passwd** (on PC-BSD) file to authenticate a user ID.

23.9.2 Access Control: Discretionary (DAC), Mandatory (MAC), and Role-Based (RBAC)

Access control policies determine what access *right* is granted to what *object*, under what circumstances (discretionary, mandatory, or role-based) and by what subject. *Discretionary access* enables an entity (like a user) to grant another entity access rights. *Mandatory access* will not allow an entity to grant another entity access rights. Rather than use the user ID to determine what access rights users and groups have on the system, the *role-based access control* (RBAC) paradigm grants access based on the role or roles that a user assumes. In ZFS, for example, when you execute the `zfs` command, your action is checked to see that the entity issuing the command has the role privilege, even if the user is root. In Solaris, the `roleadd` command and its options and arguments define roles on the system. Not only does the system administrator need to allocate login and resource use restrictions and privileges on the system, they must control all file and data access in the traditional UNIX system via the `chmod` command. Ordinarily this is done by root. On PC-BSD, you can by default use the `sudo` command as well to assume the role of root to issue the `chmod`, `chown`, or `chgrp` for files.

23.9.2.1 `sudo`

On Solaris and PC-BSD systems, programs, commands, and files are traditionally accessed through user and group permissions. Each user has a unique identifier, given either as a username or UID. Users belong to unique groups, given either as a group name or a GID. Specific users and groups have permission to access available programs, commands, and files.

The **su** command allows a user to switch roles and become the superuser on the system without logging off from their own account. You must know the password of the root account if you want to assume the role of root.

The `sudo` program allows a single command to be run as root, or even as some other user. The system administrator utilizes a policy listing file (named **sudoers**) that contains commands that each user can execute. When any user needs to run a command that requires root permissions, that user types `sudo command` in a console terminal, allowing them to run `command`. Then, `sudo` consults its permissions list in the policy listing file. If the user has permission to run that command, it runs the command. If the user does not have permission to run the command, `sudo` denies execution. Running `sudo` <u>does not</u> require knowing root's password, but by default requires the user's own password to execute successfully.

There are two aspects to `sudo`: the `sudo` program itself, and the **sudoers** policy file that the program uses. The **sudoers** policy file can only be edited by root. The `sudo` program includes a special tool, `visudo`, just for editing and validating the **sudoers** policy file. On Solaris, the path to the executable program `visudo` is **/usr/sbin**, and on PC-BSD the path to `visudo` is **/usr/local/sbin**.

The **sudoers** policy file *must* only be edited with `visudo`, because that special editing tool has safeguards built into it. On Solaris and PC-BSD `visudo` launches the vi text editor to allow editing of the **sudoers** file. On Solaris, the **sudoers** file itself is found in **/etc**. In

PC-BSD, the **sudoers** file is found in **/usr/local/etc**. At this point, you should use the more command to examine the contents of the **sudoers** file on your system.

The **sudoers** file recognizes seven types of user specification lists. They are usernames, group names (such as **wheel** on PC-BSD), aliases defined within the **sudoers** file itself, UID numbers, GID numbers, netgroups, and non-UNIX groups. See the following examples for applications of some of these seven specification lists.

The **sudoers** file is composed of two types of entries: aliases (basically variables) and user specifications that specify who may run what. When multiple entries match a user, they are applied in order. Where there are multiple matches, the last match is used (which is not necessarily the most specific match). A user specification determines which commands a user may run (and as what user) on specified hosts. By default, commands are run as root, but this can be changed on a per-command basis.

The user specifications in the **sudoers** file contain policy rules, one rule per line. Every rule uses the general format as follows:

```
who where = (as_whom) what
```

where:
who is the user that this rule applies to. who can also be a user specification list—for example, a group name that has been defined as shown for Solaris in Section 23.3.3.7, "Adding Group Accounts," and for PC-BSD in Section 23.3.2, "Adding/Deleting and Maintaining Users and Groups in a GUI-Based Interface on PC-BSD."

where is the hostname of the system this rule applies to.

= separates the where from (as _ whom) and what.

(as _ whom) designates the user specification list sudo will run the what.

what lists the full path to each command this policy rule applies to.

You must specify full pathnames to command.

Some **sudoers** file examples follow:

In the following rule, **bob** can run any command:

```
bob ALL = ALL
```

The following rule from a Solaris **sudoers** file allows user **sarwar** to run the visudo program:

```
sarwar ALL = /usr/sbin/visudo
```

The following command run on a Solaris system, uses the -l option to sudo to allow you to list the permissions currently defined as policy in the **sudoers** file. We see that user **sarwar** can run all commands as root, all commands as the user **admin**, and visudo as **root**.

```
$ sudo -l
```

User **sarwar** may run the following commands on this host:

```
(root) ALL
(admin) ALL
(root) /usr/sbin/visudo
```

This is an entry in the **sudoers** file on a Solaris system, where **dgb** can be an alias defined in the **sudoers** file that includes more than one user, specifying that the user **dgb** may run **/bin/ls**, **/bin/kill**, and **/usr/bin/lprm**—but only as the user operator on the host **solaris11_2**.

```
dgb solaris11_2 = (operator) /bin/ls, /bin/kill, /usr/bin/lprm
```

Use of a group name on Solaris, where everyone in the group **man** can run all of the commands in **/opt//bin** as the user database, on the server **solaris11_2**:

```
%man solaris11_2 = (database) /opt//bin/*
```

Use of a user ID number in a **sudoers** file to allow the user with ID **1002** on a PC-BSD system to run everything in the **/usr/local/sbin** directory:

```
#1002 ALL = /usr/local/sbin/*
```

To see more examples, and get a complete reference for the `sudo` command, use the man page for `sudo` on your system.

23.9.3 Using Access Control Lists (ACLs) in PC-BSD

In contrast to the traditional UNIX permissions model, which defines secure access to an object like a file or directory via permissions like read, write, and execute, the access control list (ACL) model gives the user base finer-grained control over object security.

In the permissions model, group permissions are the only way by which a file owner can relegate access to different constituencies of the user base. That is because any file can only belong to one group. Therefore, to serve different constituencies, many different groups have to exist. Only administrators can create and assign group membership. And sharing of files between collaborative working project teams becomes untenable using the permissions model.

ACLs provide greater discretionary power, but at the cost of more complexity, larger storage requirements, and slowing of performance of the underlying file system, whether that be VFS or ZFS. Our two UNIX base systems, PC-BSD and Solaris, both with ZFS, support the permissions model and the ACL model (referred to here as NFSv4). Since there are several ACL models, it is worth noting here that ZFS only supports NFSv4 ACLs.

We showed the permissions model in Chapter 5, Section 5.4. The methods shown in this section apply to PC-BSD ZFS files. In Chapter 24, Section 5, we show a more advanced and detailed method of managing ACLs for ZFS files and directories on a Solaris system.

We cover the basics of the following topics in the subsections indicated:

23.9.3.1: PC-BSD ACL Model

23.9.3.2: Setting ACLs on ZFS Files

23.9.3.3: Setting ACL Inheritance on ZFS Files

23.9.3.1 PC-BSD ACL Model

The PC-BSD default implementation of ACLs uses NSFv4 ACL syntax, as opposed to POSIX.1e syntax. The PC-BSD ACL model fully supports the interoperability that NFSv4 offers between UNIX and non-UNIX clients. It provides more detailed access control than is available with POSIX.1e standard file permissions. These ACLs are set and displayed with the `setfacl` and `getfacl` commands.

The ACL model has two types of *access control entries* (ACEs) that affect access checking: `ALLOW` and `DENY`. Therefore, you cannot infer from any single ACE that defines a set of permissions whether the permissions that are not defined in that ACE are allowed or denied.

23.9.3.1.1 ACL Formats

ACLs have two basic formats:

1. Trivial (Minimal) ACL contains only entries for traditional UNIX user categories that are represented as `owner@`, `group@`, and `everyone@`.

   For a newly created file, the default ACL has the following entries:

```
# file: acl_default_for_file
# owner: bob
# group: bob
          owner@:rw-p--aARWcCos:------:allow
          group@:r-----a-R-c--s:------:allow
       everyone@:r-----a-R-c--s:------:allow
```

   For a newly created directory, the default ACL has the following entries:

```
# file: acl_default_for_directories
# owner: bob
# group: bob
          owner@:rwxp--aARWcCos:------:allow
          group@:r-x---a-R-c--s:------:allow
       everyone@:r-x---a-R-c--s:------:allow
```

2. Nontrivial (Extended) ACL contains entries for added user categories. The entries might also include inheritance flags or be ordered in a nontraditional way.

A nontrivial entry might look like the following example, where permissions are specifically granted to user **mansoor**:

```
0:user:mansoor:read_data/write_data:file_inherit:allow
```

23.9.3.1.2 ACL Entry Descriptions: Components of NFSv4 ACL Command Entry Descriptions The following describes the syntactic components, both general and specific, of the `setfacl` command applied to files and directories. We also show the general form of the output from the `getfacl` command after the example is executed:

```
        Command  To Whom              Permissions       To What
Files    setfacl -a 0 user:bob:rwx---------:----:allow filename
           a     b c  d    e   f             g     h      i

Directories  setfacl -a 0 user:bob:r-------:fd---:allow dirname
                  a     b c   d   e  f         g     h      i
key-
a-command
b-option
c-position option argument starting at 0, cannot be a number at
the end!
d- ACL tag
e- ACL qualifier, in these cases user
f- 14 permissions, in short form, the hyphens(-) optional
g- 6 inheritance flags, for directories only, the hyphens(-)
optional
h- ACL type, either allow or deny
i- command argument, a filename or directory name specification
```

Format of `getfacl` output for the preceding file's command:

```
%getfacl filename
#file         filename
#owner     owner name
#group      group name
position 0             user:bob:rwx-----------:------:allow
filename
position1          owner@--------------------------------------
position2          group@--------------------------------------
position3          everyone@-----------------------------------
```

Use the following additional sample entry as a reference to the elements that comprise an ACL entry. These elements apply to both trivial and nontrivial ACLs.

```
user:mansoor:read_data/write_data:file_inherit:allow
```

ACL tag and qualifier: The user category. In trivial ACLs, only entries for `owner@`, `group@`, and `everyone@` are set. In nontrivial ACLs, `user:username` and `group:groupname` are added. In the example, the entry tag and qualifier are `user:mansoor`.

Access privileges: Permissions that are granted or denied to the entry type. In the example, user **mansoor**'s permissions are shown in long form as `read _ data` and `write _ data`.

Inheritance flags: An optional list of ACL flags that control how permissions are propagated in a directory structure. In the sample entry, `file _ inherit` is also granted to user **mansoor**.

ACL type (permission control): Determines whether the permissions in an entry are allowed or denied. In the example, the permissions for **mansoor** are allowed.

Table 23.11 describes ACL forms, and Table 23.12 describes default ACL entries. Table 23.13 describes each ACL entry type more fully. Table 23.14 describes ACL access privileges more fully.

TABLE 23.11 ACL Forms

ACL Entry	Description
`u[ser]::perms`	File owner permissions.
`g[roup]::perms`	File group permissions.
`o[ther]:perms`	Permissions for users other than the file owner or members of file group.
`m[ask]:perms`	The ACL mask. The mask entry indicates the maximum permissions allowed for users (other than the owner) and for groups. The mask is a quick way to change permissions on all the users and groups. For example, the `mask:r--` mask entry indicates that users and groups cannot have more than read permissions, even though they might have write/execute permissions.
`u[ser]:uid:perms`	Permissions for a specific user. For `uid`, you can specify either a user name or a numeric UID.
`g[roup]:gid:perms`	Permissions for a specific group. For `gid`, you can specify either a group name or a numeric GID.

TABLE 23.12 Default ACL Entries

Default ACL Entry	Description
`d[efault]:u[ser]::perms`	Default file owner permissions.
`d[efault]:g[roup]::perms`	Default file group permissions.
`d[efault]:o[ther]:perms`	Default permissions for users other than the file owner or members of the file group.
`d[efault]:m[ask]:perms`	Default ACL mask.
`d[efault]:u[ser]:uid:perms`	Default permissions for a specific user. For `uid`, you can specify either a user name or a numeric UID.
`d[efault]:g[roup]:gid:perms`	Default permissions for a specific group. For `gid`, you can specify either a group name or a numeric GID.

TABLE 23.13 Generic ACL Entry Types

ACL Entry Type	Description
`owner@`	Specifies the access granted to the owner of the object
`group@`	Specifies the access granted to the owning group of the object
`everyone@`	Specifies the access granted to any user or group that does not match any other ACL entry
`user`	With a user name, specifies the access granted to an additional user of the object
`group`	With a group name, specifies the access granted to an additional group of the object

TABLE 23.14 Generic ACL Access Privileges

Access Privilege	Compact Access Privilege	Description
`add_file`	`w`	Permission to add a new file to a directory
`add_subdirectory`	`p`	On a directory, permission to create a subdirectory
`append_data`	`p`	Permission to modify a file but only beginning from the EOF
`delete`	`d`	Permission to delete a file
`delete_child`	`D`	Permission to delete a file or directory within a directory
`execute`	`x`	Permission to execute a file or search the contents of a directory
`list_directory`	`r`	Permission to list the contents of a directory
`read_acl`	`c`	Permission to read the ACL (`ls`)
`read_attributes`	`a`	Permission to read basic attributes (non-ACLs) of a file
`read_data`	`r`	Permission to read the contents of the file
`read_xattr`	`R`	Permission to read the extended attributes of a file
`synchronize`	`s`	Permission to access a file locally at the server with synchronized read and write operations
`write_xattr`	`W`	Permission to create extended attributes or write to the extended attributes directory
`write_data`	`w`	Permission to modify or replace the contents of a file
`write_attributes`	`A`	Permission to change the times associated with a file or directory to an arbitrary value
`write_acl`	`C`	Permission to write the ACL or the ability to modify the ACL by using the chmod command
`write_owner`	`o`	Permission to change the file's owner or group, or the ability to execute the `chown` or `chgrp` commands on the file

Table 23.15 provides additional details about ACL `delete` and `delete _ child` behavior.

23.9.3.1.2.1 ZFS ACL Sets An ACL set consists of a combination of ACL permissions. These ACL sets are predefined and cannot be modified.

ACL Set Name	Included ACL Permissions
`full _ set`	All permissions

TABLE 23.15 ACL `delete` and `delete_child` Permission Behavior

Parent Directory Permissions	Target Object Permissions		
" " (empty)	**ACL Allows Delete**	**ACL Denies Delete**	**Delete Permission Unspecified**
ACL allows `delete_child`	Permit	Permit	Permit
ACL denies `delete_child`	Permit	Deny	Deny
ACL allows only write and execute	Permit	Permit	Permit
ACL denies write and execute	Permit	Deny	Deny

`modify _ set`	All permissions except `write _ acl` and `write _ owner`
`read _ set`	`read _ data`, `read _ attributes`, `read _ xattr`, `read _ acl`
`write _ set`	`write _ data`, `append _ data`, `write _ attributes`, `write _ xattr`

You can apply an ACL set rather than having to set individual permissions separately. The advantage of using these predefined sets of ACLs is that you can apply a blanket grouping of ACLs to a file or a directory all at one time. In the following example, granting Mansoor the `read _ set` ACL set gives him permissions to read ACLs as well as file contents and their basic and extended attributes for **file.0**:

```
% setfacl -m user:mansoor:read_set:allow file.0
% getfacl file.0
# file: file.0
# owner: bob
# group: bob
    user:mansoor:r-----a-R-c---:------:allow
          owner@:rw-p--aARWcCos:------:allow
          group@:r-----a-R-c--s:------:allow
       everyone@:r-x---a-R-c--s:------:allow
```

23.9.3.1.3 ACL Inheritance ACL inheritance means that a newly created file or directory can inherit the ACLs they are intended to inherit without disregarding the existing permission bits on the parent directory. By default, ACLs are not propagated. If you set a nontrivial ACL on a directory, it is not inherited to any subsequent directory. You must specify the inheritance of an ACL on a file or directory.

Table 23.16 describes the optional inheritance flags.

In addition, you can set a default ACL inheritance policy on the file system that is more or less strict by using the `aclinherit` file system property. For more information about this property, see Section 23.9.3.1.4, "ACL Properties."

TABLE 23.16 Generic ACL Inheritance Flags

Inheritance Flag	Compact Inheritance Flag	Description
`file_inherit`	`f`	Only inherit the ACL from the parent directory to the directory's files
`dir_inherit`	`d`	Only inherit the ACL from the parent directory to the directory's subdirectories
`inherit_only`	`I`	Inherit the ACL from the parent directory
`no_propagate`	`n`	Only inherit the ACL from the parent directory to the first-level contents of the directory
`-`	n/a	No permission granted
`successful_access`	`S`	Indicates whether an alarm or audit record should be initiated upon a successful access; used with audit or alarm ACE types
`failed_access`	`F`	Indicates whether an alarm or audit record should be initiated when an access fails; used with audit or alarm ACE types
`inherited`	I	Indicates that an ACE was inherited

For more information about setting ACL inheritance on ZFS files, see Section 23.9.3.2, "Setting ACL Inheritance on ZFS Files."

23.9.3.1.4 ACL Properties The ZFS file system includes the ACL properties to determine the specific behavior of ACL inheritance and ACL interaction with `setfacl` command operations.

These properties are

- `aclinherit`: Determines the behavior of ACL inheritance. Values include the following:

  `discard`: For new objects, no ACL entries are inherited when a file or directory is created. The ACL on the file or directory is equal to the permission mode of the file or directory.

  `noallow`: For new objects, only inheritable ACL entries that have an access type of `deny` are inherited.

  `restricted`: For new objects, the `write _ owner` and `write _ acl` permissions are removed when an ACL entry is inherited.

  `passthrough`: When a property value is set to `passthrough`, files are created with a mode determined by the inheritable ACEs. If no inheritable ACEs exist that affect the mode, then the mode is set in accordance to the requested mode from the application.

  `passthrough-x`: Has the same semantics as `passthrough` except that when `passthrough-x` is enabled, files are created with the execute (`x`) permission only if the execute permission is set in file creation mode and in an inheritable ACE that affects the mode.

The default mode for the `aclinherit` is restricted.

For more information about the `aclinherit` modes, see Section 23.9.3.3.3, "Modifying ACL Inheritance with the ACL Inherit Mode."

- `aclmode`: Modifies ACL behavior when a file is initially created or controls how an ACL is modified during a `chmod` operation. Values include the following:

  `discard`: Deletes all ACL entries that do not represent the mode of the file. This is the default value.

  `mask`: Reduces user or group permissions. The permissions are reduced such that they are no greater than the group permission bits unless it is a user entry that has the same UID as the owner of the file or directory. In this case, the ACL permissions are reduced so that they are no greater than owner permission bits. The `mask` value also preserves the ACL across mode changes, provided that an explicit ACL set operation has not been performed.

  `passthrough`: Indicates that no changes are made to the ACL other than generating the necessary ACL entries to represent the new mode of the file or directory.

The default mode for the `aclmode` is `discard`.

23.9.3.2 Setting ACLs on ZFS Files

The primary rules of ACL access on a ZFS file are as follows.

1. ZFS processes ACL entries in the order they are listed in the ACL, from the top down.
2. Only ACL entries whose specified user matches the requester of the access are processed.
3. Once an allow permission has been granted, it cannot be denied by a subsequent ACL deny entry in the same ACL permission set.
4. The owner of the file is granted the `write _ acl` permission unconditionally even if the permission is explicitly denied. Otherwise, any permission left unspecified is denied.

In the cases of deny permissions or when an access permission is missing, the privilege subsystem determines the access request that is granted for the owner of the file or for superuser. This mechanism prevents owners of files from getting locked out of their files and enables superuser to modify files for recovery purposes.

23.9.3.2.1 Command Syntax for Setting and Viewing ACLs To set or modify ACLs, use the `setfacl` command. To see the results of using `setfacl`, use the `getfacl` command. Following are an abbreviated syntax description of those two commands, with common and allowable NFSv4 options shown:

SYNTAX

```
setfacl [-bdhkn] [-a position entries] [-m entries] [-M file] [-x entries
| position] [-X file] [file ...]
```

Purpose:

The **setfacl** utility sets discretionary access control information on the specified file(s).

Output: Modified ACL specifications

Common Options and Option Arguments:

-a [position] [entries] Modify the ACL on the specified files by inserting new ACL entries specified in entries, starting at **[position]**, counting from zero.

-b Remove all ACL entries except for or six *canonical* entries (NFSv4 ACLs).

-m [entries] Modify the ACL on the specified file. New entries will be added, and existing entries will be modified according to the entries argument. For NFSv4 ACLs, you can also use the **-a** and **-x** options instead.

-M file Modify the ACL entries on the specified files by adding new ACL entries and modifying existing ACL entries with the ACL entries specified in **file**. If **file** is **-**, the input is taken from **stdin**.

-x entries | position If **entries** is specified, remove the ACL entries specified there from the access or default ACL of the specified files. Otherwise, remove entry at index **position**, counting from zero.

SYNTAX

```
getfacl [-dhinqv] [file ...]
```

Purpose:

The **getfacl** utility writes discretionary access control information associated with the specified file(s) to standard output.

Output: Indicated file ACL settings on **stdout**

Common Options:

-i Append numerical ID at the end of each entry containing user or group name.

-n Display user and group IDs numerically rather than converting to a user or group name.

-q Do not write commented information about file name and ownership. This is useful when dealing with filenames with unprintable characters.

-v Display access mask and flags in a verbose form.

In the following command line example, the default ACL entry for **file.0** is modified so that the C permission is granted to `everyone@`:

```
% setfacl -m everyone@:r-----a-R-cC-s:------:allow file.0
```

Permissions and inheritance flags are represented by the unique letters listed in Table 24.2 and Table 24.4.

The following is another example of modifying the ACL entries, which gives `rx` privileges to user mansoor on **file.0**

```
% setfacl -m user:mansoor:rx:allow file.0
```

To grant user **mansoor** inheritable read, write, and execute permissions for the newly created directory **d.2** and its files, you can use the following command:

```
% setfacl -a 0 user:mansoor:read_set/execute:file_inherit/dir_
inherit:allow d.2
```

Notice that the syntax of this command uses the `-a` option of `setfacl`, and the option argument for position is `0`. This means that the new entry will occupy the first position in the entry list. Also notice the permissions format is the long form of access permissions.

23.9.3.2.1.1 Displaying ACL Information With the `getfacl` command, you can display ACL information in one of two formats. The following example shows how the same ACL information is displayed first in verbose (short) form and then in compact (short) form:

```
% getfacl -v file.0
# file: file.0
# owner: bob
# group: bob
          user:bob:read_data/execute::allow
owner@:read_data/write_data/append_data/read_attributes/write_
attributes/read_xattr/write_xattr/read_acl/write_acl/write_owner/
synchronize::allow
group@:read_data/read_attributes/read_xattr/read_acl/
synchronize::allow
everyone@:read_data/read_attributes/read_xattr/read_acl/write_acl/
synchronize::allow
% getfacl file.0
# file: file.0
# owner: bob
# group: bob
          user:bob:r-x-----------:------:allow
            owner@:rw-p--aARWcCos:------:allow
            group@:r-----a-R-c--s:------:allow
         everyone@:r-----a-R-cC-s:------:allow
```

For an explanation of the permissions for each user category, see Table 23.14.

23.9.3.2.2 Modifying ACLs on ZFS Files The following are some basic ACL modifications on PC-BSD ZFS files and directories.

Please note that on your system, in place of the user **mansoor**, you must substitute another valid username.

1. To begin, create a file in your home directory named **file.1**.

```
% touch file.1
%
```

2. Delete the `everyone@` entry in the ACLs for **file.1**:

```
% setfacl -x everyone@:r-----a-R-cC-s:------:allow file.1
% getfacl file.1
# file: file.1
# owner: bob
# group: bob
            owner@:rw-p--aARWcCos:------:allow
            group@:r-----a-R-c--s:------:allow
```

3. Add a new `everyone@` entry. Notice the `-m` option places the new entry at position 0:

```
% setfacl -m everyone@:r-----a-R-c--s:------:allow file.1
% getfacl file.1
# file: file.1
# owner: bob
# group: bob
         everyone@:r-----a-R-c--s:------:allow
            owner@:rw-p--aARWcCos:------:allow
            group@:r-----a-R-c--s:------:allow
```

4. Delete **file.1** and create a new version of it with the default ACLs:

```
% rm file.1
% touch file.1
% getfacl file.1
# file: file.1
# owner: bob
# group: bob
            owner@:rw-p--aARWcCos:------:allow
            group@:r-----a-R-c--s:------:allow
         everyone@:r-----a-R-c--s:------:allow
```

5. Modify the position 2 `everyone@` entry to `rw` only:

```
% setfacl -m everyone@:rw:allow file.1
% getfacl file.1
# file: file.1
# owner: bob
# group: bob
```

```
            owner@:rw-p--aARWcCos:------:allow
            group@:r-----a-R-c--s:------:allow
         everyone@:rw------------:------:allow
```

6. Add an ACL entry to deny user **mansoor** the read _ set privileges on **file.1**:

```
% setfacl -m user:mansoor:read_set:deny file.1
% getfacl file.1
# file: file.1
# owner: bob
# group: bob
      user:mansoor:r-----a-R-c---:------:deny
            owner@:rw-p--aARWcCos:------:allow
            group@:r-----a-R-c--s:------:allow
         everyone@:rw------------:------:allow
Notice what predefined permissions are included in the
read_set!
```

7. Add an ACL entry to deny everyone@ the read _ set privileges on **file.1:**

```
% setfacl -m everyone@:read_set:deny file.1

% getfacl file.1
# file: file.1
# owner: bob
# group: bob
         everyone@:r-----a-R-c---:------:deny
      user:mansoor:r-----a-R-c---:------:deny
            owner@:rw-p--aARWcCos:------:allow
            group@:r-----a-R-c--s:------:allow
         everyone@:rw------------:------:allow
```

8. Delete the ACL entry to deny everyone@ the read _ set privileges:

```
 % setfacl -x everyone@:read_set:deny file.1
% getfacl file.1
# file: file.1
# owner: bob
# group: bob
      user:mansoor:r-----a-R-c---:------:deny
            owner@:rw-p--aARWcCos:------:allow
            group@:r-----a-R-c--s:------:allow
```

9. Add the following privileges to everyone@ for **file.1**:

```
% setfacl -m everyone@:r-----a-R-c--s:------:allow file.1
% getfacl file.1
# file: file.1
# owner: bob
# group: bob
         everyone@:r-----a-R-c--s:------:allow
      user:mansoor:r-----a-R-c---:------:deny
            owner@:rw-p--aARWcCos:------:allow
            group@:r-----a-R-c--s:------:allow
```

Another way to accomplish the `setfacl` command would be:
% **setfacl -a 0 everyone@:r-----a-R-c--s:------:allow file.1**

10. Delete the 0 position ACL entry for **file.1**:

```
% setfacl -x 0 file.1
% getfacl file.1
# file: file.1
# owner: bob
# group: bob
     user:mansoor:r-----a-R-c---:------:deny
           owner@:rw-p--aARWcCos:------:allow
           group@:r-----a-R-c--s:------:allow
```

11. Add a `read _ set` of permissions for `everybody@` for **file.1**:

```
% setfacl -a 0 everyone@:read_set:allow file.1
% getfacl file.1
# file: file.1
# owner: bob
# group: bob
        everyone@:r-----a-R-c---:------:allow
     user:mansoor:r-----a-R-c---:------:deny
           owner@:rw-p--aARWcCos:------:allow
           group@:r-----a-R-c--s:------:allow
```

12. Delete the 0 position ACL entry for **file.1**:

```
% setfacl -x0 file.1
% getfacl file.1
# file: file.1
# owner: bob
# group: bob
     user:mansoor:r-----a-R-c---:------:deny
           owner@:rw-p--aARWcCos:------:allow
           group@:r-----a-R-c--s:------:allow
```

13. Add the `read _ xattr` permission for user **mansoor** to **file.1**:

```
% setfacl -m user:mansoor:read_xattr::allow file.1
% getfacl file.1
# file: file.1
# owner: bob
# group: bob
      user:mansoor:--------R-----:------:allow
      user:mansoor:r-----a-R-c---:------:deny
            owner@:rw-p--aARWcCos:------:allow
            group@:r-----a-R-c--s:------:allow
read_
After these commands are executed, does user mansoor have
read_xattr permission on file.1?
```

14. Delete `read _ xattr` permission for user **mansoor** on **file.1**:

```
% setfacl -x user:mansoor:read_xattr::allow file.1
% getfacl file.1
# file: file.1
# owner: bob
# group: bob
      user:mansoor:r-----a-R-c---:------:deny
            owner@:rw-p--aARWcCos:------:allow
            group@:r-----a-R-c--s:------:allow
```

15. Add `r-----a-R-c---:------` privileges for user **mansoor** on **file.1**:

```
% setfacl -a 0 user:mansoor:r-----a-R-c---:------:allow file.1
% getfacl file.1
# file: file.1
# owner: bob
# group: bob
      user:mansoor:r-----a-R-c---:------:allow
      user:mansoor:r-----a-R-c---:------:deny
            owner@:rw-p--aARWcCos:------:allow
            group@:r-----a-R-c--s:------:allow
```

16. Delete position 1 ACL entry for **file.1**:

```
% setfacl -x 1 file.1
% getfacl file.1
# file: file.1
# owner: bob
# group: bob
      user:mansoor:r-----a-R-c---:------:allow
            owner@:rw-p--aARWcCos:------:allow
            group@:r-----a-R-c--s:------:allow
```

17. Delete position 0 ACL entry for **file.1**:

```
% setfacl -x 0 file.1
% getfacl file.1
# file: file.1
# owner: bob
# group: bob
            owner@:rw-p--aARWcCos:------:allow
            group@:r-----a-R-c--s:------:allow
```

18. Add position 0 entry of `r-----a-R-c---:------` ACL for user **mansoor** on **file.1**:

```
% setfacl -a 0 user:mansoor:r-----a-R-c---:------:allow file.1
% getfacl file.1
# file: file.1
# owner: bob
# group: bob
      user:mansoor:r-----a-R-c---:------:allow
            owner@:rw-p--aARWcCos:------:allow
            group@:r-----a-R-c--s:------:allow
```

19. Add position 0 entry of the `read _ set` for `everyone@` on **file.1**:

```
% setfacl -a 0 everyone@:read_set:allow file.1
% getfacl file.1
# file: file.1
# owner: bob
# group: bob
         everyone@:r-----a-R-c---:------:allow
      user:mansoor:r-----a-R-c---:------:allow
            owner@:rw-p--aARWcCos:------:allow
            group@:r-----a-R-c--s:------:allow
```

The following steps deal with PC-BSD directories.

20. Create a new directory, under your home directory, named **dir_acl**, and view its default ACL entries and permissions:

```
% mkdir dir_acl
% getfacl dir_acl
# file: dir_acl
# owner: bob
# group: bob
            owner@:rwxp--aARWcCos:------:allow
            group@:r-x---a-R-c--s:------:allow
         everyone@:r-x---a-R-c--s:------:allow
```

21. Add `r-----a-R-c---:------` permissions to user **mansoor** for the directory **dir_acl**:

```
% setfacl -a 0 user:mansoor:r-----a-R-c---:------:allow
dir_acl
% getfacl dir_acl
# file: dir_acl
# owner: bob
# group: bob
      user:mansoor:r-----a-R-c---:------:allow
            owner@:rwxp--aARWcCos:------:allow
            group@:r-x---a-R-c--s:------:allow
         everyone@:r-x---a-R-c--s:------:allow
```

22. Add some additional permissions at position 0 for user **mansoor** on directory **dir_acl**:

```
% setfacl -a 0 user:mansoor:read_set/execute:file_inherit/
dir_inherit:allow dir_acl
% getfacl dir_acl
# file: dir_acl
# owner: bob
# group: bob
      user:mansoor:r-x---a-R-c---:fd----:allow
      user:mansoor:r-----a-R-c---:------:allow
            owner@:rwxp--aARWcCos:------:allow
            group@:r-x---a-R-c--s:------:allow
         everyone@:r-x---a-R-c--s:------:allow
```

23. Delete the position 1 ACL entry for **dir_acl**:

```
% setfacl -x 1 dir_acl
% getfacl dir_acl
# file: dir_acl
# owner: bob
# group: bob
      user:mansoor:r-x---a-R-c---:fd----:allow
            owner@:rwxp--aARWcCos:------:allow
            group@:r-x---a-R-c--s:------:allow
         everyone@:r-x---a-R-c--s:------:allow
```

24. Delete the position 0 ACL entry for **dir_acl**:

```
% setfacl -x 0 dir_acl
% getfacl dir_acl
# file: dir_acl
# owner: bob
# group: bob
            owner@:rwxp--aARWcCos:------:allow
```

```
             group@:r-x---a-R-c--s:------:allow
          everyone@:r-x---a-R-c--s:------:allow
```

25. Add long-form permissions for user **bob** on **dir_acl**:

```
% setfacl -a 0 user:bob:read_data/write_data/execute:file_
inherit/dir_inherit:allow / dir_acl
% getfacl dir_acl
# file: dir_acl
# owner: bob
# group: bob
           user:bob:rwx-----------:fd----:allow
             owner@:rwxp--aARWcCos:------:allow
             group@:r-x---a-R-c--s:------:allow
          everyone@:r-x---a-R-c--s:------:allow
```

26. Create a new file in the directory **dir_acl** named **file.2**:

```
% touch dir_acl/file.2
% getfacl dir_acl/file.2
# file: dir_acl/file.2
# owner: bob
# group: bob
           user:bob:r-------------:------:allow
             owner@:rw-p--aARWcCos:------:allow
             group@:r-----a-R-c--s:------:allow
          everyone@:r-----a-R-c--s:------:allow
```

Why did **file.2** only inherit `r` privilege for user **bob**?

EXERCISE 23.17

Execute all 26 of the preceding command line examples and verify that the output of each is the same on your PC-BSD system.

23.9.3.3 Setting ACL Inheritance on ZFS Files

You can determine how ACLs are inherited on files and directories.

The `aclinherit` property can be set globally on a file system, and in our examples we are working on the file system **tank/usr/home/bob**. Our default `aclinherit` mode was set at `discard`. Because the `aclinherit` property for any file system is set to some default mode, do the following to ascertain the `aclinherit` mode, and then set it to `inherit`. You must change the dataset name from **tank/usr/home/bob** to the whatever ZFS dataset you are working in on your system, and execute the `zfs set` command as superuser.

```
% zfs get aclinherit tank/usr/home/bob
NAME               PROPERTY    VALUE         SOURCE
tank/usr/home/bob  aclinherit  noallow       local
% su
Password: xxx
# zfs set aclinherit=restricted tank/usr/home/bob
# exit
Exit
```

For more information, see Section 23.9.3.1.3, "ACL Inheritance."

23.9.3.3.1 Granting ACLs that Are Inherited by Files This section identifies the file ACEs that are applied when the `file _ inherit` flag is set.

In the following example, `read _ data/write _ data` permissions are added for files in the **test2.dir** directory for user **bob** so that he has read access on any newly created files.

```
% mkdir test2.dir
% setfacl -m user:bob:read_data/write_data:file_inherit:allow
test2.dir
% getfacl test2.dir
# file: test2.dir
# owner: bob
# group: bob
          user:bob:rw------------:f-----:allow
            owner@:rwxp--aARWcCos:------:allow
            group@:r-x---a-R-c--s:------:allow
         everyone@:r-x---a-R-c--s:------:allow
```

In the following example, user **bob**'s permissions are applied on the newly created **test2.dir/file.2** file. The ACL inheritance granted, `read _ data:file _ inherit:allow`, means user **bob** can read the contents of any newly created file.

```
% touch test2.dir/file.2
% getfacl test2.dir/file.2
# file: test2.dir/file.2
# owner: bob
# group: bob
          user:bob:r-------------:------:allow
            owner@:rw-p--aARWcCos:------:allow
            group@:r-----a-R-c--s:------:allow
         everyone@:r-----a-R-c--s:------:allow
```

Thus, user **bob** does not have `write _ data` permission on **file.2** because the group permission of the file does not allow it.

The `inherit _ only` permission, which is applied when the `file _ inherit` or `dir _ inherit` flags are set, is used to propagate the ACL through the directory structure. As such, user **bob** is granted or denied permission from `everyone@` permissions only if he is the file owner or is a member of the file's group owner. For example:

```
% mkdir test2.dir/subdir.2
% getfacl test2.dir/subdir.2
# file: test2.dir/subdir.2
# owner: bob
# group: bob
          user:bob:rw-----------:f-i---:allow
            owner@:rwxp--aARWcCos:------:allow
            group@:r-x---a-R-c--s:------:allow
         everyone@:r-x---a-R-c--s:------:allow
```

23.9.3.3.2 Granting ACLs that Are Inherited by Both Files and Directories This section provides examples that identify the file and directory ACLs that are applied when both the `file _ inherit` and `dir _ inherit` flags are set.

In the following example, user **bob** is granted read, write, and execute permissions that are inherited for newly created files and directories.

```
% mkdir test3.dir
% setfacl -m user:bob:read_data/write_data/execute:file_inherit/
dir_inherit:allow / test3.dir
% getfacl test3.dir
# file: test3.dir
# owner: bob
# group: bob
            user:bob:rwx-----------:fd----:allow
            owner@:rwxp--aARWcCos:------:allow
            group@:r-x---a-R-c--s:------:allow
         everyone@:r-x---a-R-c--s:------:allow
```

The inherited text in the following output is informational, and indicates that the ACE is inherited.

```
% touch test3.dir/file.3
% getfacl test3.dir/file.3
# file: test3.dir/file.3
# owner: bob
# group: bob
           user:bob:r-------------:------:allow
            owner@:rw-p--aARWcCos:------:allow
            group@:r-----a-R-c--s:------:allow
```

```
         everyone@:r-----a-R-c--s:------:allow
```

In these examples, because the permission bits of the parent directory for `group@` and `everyone@` deny write and execute permissions, user **bob** is denied write and execute permissions. The default `aclinherit` property is restricted, which means that `write _ data` and execute permissions are not inherited.

In the following example, user **bob** is granted read, write, and execute permissions that are inherited for newly created files, but are not propagated to subsequent contents of the directory.

```
% mkdir test4.dir
% setfacl -m user:bob:read_data/write_data/execute:file_inherit/
no_propagate:allow / test4.dir
% getfacl test4.dir
# file: test4.dir
# owner: bob
# group: bob
            user:bob:rwx-----------:f--n--:allow
            owner@:rwxp--aARWcCos:------:allow
            group@:r-x---a-R-c--s:------:allow
         everyone@:r-x---a-R-c--s:------:allow
```

As the following example illustrates, **bob's** `read _ data/write _ data/execute` permissions are reduced based on the owning group's permissions.

```
% touch test4.dir/file.4
% getfacl test4.dir/file.4
# file: test4.dir/file.4
# owner: bob
# group: bob
          user:bob:r-------------:------:allow
            owner@:rw-p--aARWcCos:------:allow
            group@:r-----a-R-c--s:------:allow
         everyone@:r-----a-R-c--s:------:allow
```

23.9.3.3.3 Modifying ACL Inheritance with the ACL Inherit Mode This section describes the `aclinherit` property values, and the following command line examples illustrate ACL inheritance with the ACL inherit mode set to discard. If the ZFS `aclinherit` property on a file system is set to discard, then ACLs can potentially be discarded when the permission bits on a directory change.

You must change the dataset name from **tank/usr/home/bob** to the whatever ZFS dataset you are working in on your system, and execute the `zfs set` command as superuser. Also, our PC-BSD system already had the `aclinherit` mode set to `discard`, but we presume that yours has not.

```
# zfs set aclinherit=discard tank/usr/home/bob
% mkdir test5.dir
% setfacl -m user:bob:read_data/write_data/execute:dir_
inherit:allow test5.dir
% getfacl test5.dir
# file: test5.dir
# owner: bob
# group: bob
          user:bob:rwx-----------:-d----:allow
            owner@:rwxp--aARWcCos:------:allow
            group@:r-x---a-R-c--s:------:allow
         everyone@:r-x---a-R-c--s:------:allow
```

If, at a later time, you decide to tighten the permission bits on a directory, the nontrivial ACL is discarded. For example:

```
% chmod 744 test5.dir
% getfacl test5.dir
# file: test5.dir
# owner: bob
# group: bob
            owner@:rwxp--aARWcCos:------:allow
            group@:r-----a-R-c--s:------:allow
         everyone@:r-----a-R-c--s:------:allow
```

Finally, we will illustrate ACL inheritance with the ACL inherit mode set to `noallow`.

In the following example, two nontrivial ACLs with file inheritance are set. One ACL allows `read _ data` permission, and one ACL denies `read _ data` permission. Again, you must change the dataset name from **tank/usr/home/bob** to the whatever ZFS dataset you are working in on your system, and execute the `zfs set` command as superuser.

```
# zfs set aclinherit=noallow tank/usr/home/bob
% mkdir test6.dir
% setfacl -m user:bob:read_data:file_inherit:deny test6.dir
% setfacl -m user:mansoor:read_data:file_inherit:allow test6.dir
% getfacl test6.dir
# file: test6.dir
# owner: bob
# group: bob
            user:mansoor:r-------------:f-----:allow
            user:bob:r-------------:f-----:deny
            owner@:rwxp--aARWcCos:------:allow
            group@:r-x---a-R-c--s:------:allow
         everyone@:r-x---a-R-c--s:------:allow
```

As the following example shows, when a new file is created, the ACL that allows `read _ data` permission to user **mansoor** is discarded.

```
% touch test6.dir/file.6
% getfacl test6.dir/file.6
# file: test6.dir/file.6
# owner: bob
# group: bob
            user:bob:r-------------:------:deny
            owner@:rw-p--aARWcCos:------:allow
            group@:r-----a-R-c--s:------:allow
         everyone@:r-----a-R-c--s:------:allow
```

EXERCISE 23.18

Execute the all the command line examples shown in Sections 23.9.3.3.1 through 23.9.3.3.3 on your PC-BSD system and verify that they give the same output as shown.

23.9.4 Intrusion Detection and Intrusion Detection Systems

Intrusion detection is usually applied via a software system, and most importantly through log file monitoring. Consult online sources for a more complete description of the interplay of both. A conceptual layout of how malicious activity from outside of the operating system interfaces through the components of a UNIX system in given in Figure 23.21.

23.9.5 System Firewall

A firewall is a facility that prevents unauthorized access to or from a private network or a computer. Firewalls can be implemented in both hardware and software, or a combination of both. Firewalls are primarily used to prevent unauthorized access to a private network or intranet from the Internet. All traffic entering or leaving a single computer or intranet passes through the firewall, which examines each message and blocks those that do not meet the *firewall rules*.

Hardware firewalls can be purchased as stand-alone products; they can also be found integral to broadband routers, such as the Actiontec PK5000. Most hardware firewalls will have a minimum of four network ports to connect other computers, but for larger networks, business networking firewall solutions are available.

For our two base UNIX systems, the salient difference, with respect to the default implementation of a software-implemented firewall, is as follows:

In PC-BSD, all incoming traffic is *blocked* until you specify a TCP or UDP port that traffic is allowed to come in on, and in Solaris all incoming traffic is allowed in. On both systems, all outgoing traffic on all ports is allowed. In the next section we show the specific details of how to *unblock* a port in PC-BSD.

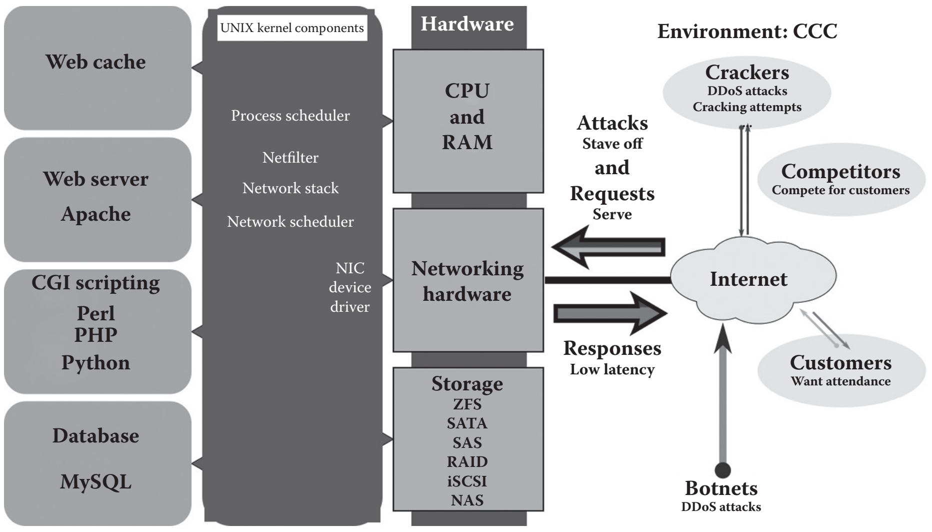


FIGURE 23.21 Routes of attack and system components.

Solaris systems have a graphical firewall manager, accessed from the **System>Administration>System Firewall** menu choice. We do not show how to establish firewall rules or manage the firewall using the default firewall manager in Solaris.

23.9.5.1 Firewall Manager in PC-BSD

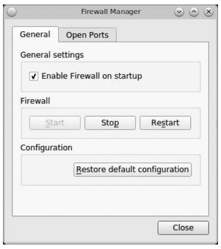
PC-BSD uses an IPFW firewall to protect your system. By default, the firewall is configured to allow all outgoing connections, but to deny all incoming connection requests. The default rulebase is located in **/etc/ipfw.rules**. Instead of showing the text-based method of modifying firewall rules, we use the Firewall Manager GUI utility to view and modify the existing firewall rules. If you want to know more about text-based modifications to firewall rules, see the `ipfw` man page on your PC-BSD system.

It is not absolutely necessary to change the firewall rules, and the details of using a command line method of doing this are not shown here. Be aware that adding custom rules or modifying the firewall jeopardizes your system's security.

To access the Firewall Manager, go to **Control Panel>Firewall Manager** or type `pc-su pc-fwmanager`. You will be prompted to input your password. Figure 23.22 shows the initial screen when you launch this utility.

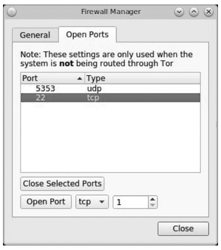
The `General` tab of this utility allows you to:

1. Determine whether or not the firewall starts when the system boots. Unless you have a reason to do so and understand the security implications, the `Enable Firewall on startup` box should be checked so that your system is protected by the firewall.
2. **Start**, **Stop**, or **Restart** the firewall.



[✓] Enable Firewall on startup

FIGURE 23.22 Firewall Manager utility.



Note: These settings are only used when the system is **not** being routed through Tor

```
Port      Type
5353      udp
22        tcp
```

tcp 1

FIGURE 23.23 Adding a new firewall rule.

3. The `Restore Default Configuration` button allows you to return to the original, working configuration.

4. To add or delete custom firewall rules, click the `Open Ports` tab to open the screen shown in Figure 23.23. Note that your custom rules will allow incoming connections on the specified protocol and port number.

Any rules that you create will appear in this screen. To add a rule, input the port number to open. By default, TCP is selected. If the rule is for the UDP protocol, click the **TCP** drop-down menu and select **UDP**. Once you have the protocol and port number selected, click the `Open Port` button to add the new rule to your custom list. Figure 23.23 shows the addition of a new rule allowing a TCP connection on port 22, the common default for ssh.

If you have created any custom rules and wish to delete one, highlight the rule to delete and click the `Close Selected Ports` button to remove it from the custom rules list.

Whenever you add or delete a custom rule, the rule will not be used until you click the `Restart` button shown in Figure 23.22. Whenever you create a custom rule, test that your new rule works as expected. As shown in Figure 23.32, if you create a rule to allow incoming ssh connections, try connecting to your PC-BSD system using ssh to verify that the firewall is now allowing the connection.

26.9.5.2 PF

A kernel-level packet filter software system screens network packets by checking the properties of individual packets and the network connections built from those packets against the filtering rules defined in its rule configuration files. The packet filter arbitrates the disposition of those packets. This could mean passing them through or rejecting them, or it could trigger events that parts of the operating system or external applications work on to dispose of the packets.

The OpenBSD UNIX implementation of packet filtering, named PF, is available in the base FreeBSD, PC-BSD systems, and Solaris. PF lets you write filtering rules to control network traffic based on essentially any packet or connection property, including address family, source and destination address, interface, protocol, port, and direction.

A packet filter can keep unwanted traffic out of your individual computer or network-connected computers. It can also help keep network traffic inside your own network. Both these functions are important to the firewall concept, but blocking is not the only useful feature of a packet filter.

Filtering can also be used to direct specific network traffic to specific hosts, assign classes of traffic to queues, perform traffic *shaping*, and dispose of selected kinds of traffic to other software for special treatment.

All this processing happens at the network level, based on packet and connection properties. PF is part of the network stack, firmly embedded in the operating system kernel. The filtering is performed in the kernel because of performance considerations. One of the most important and common actions that PF performs is to block traffic.

PF on PC-BSD

To enable PF on PC-BSD, do the following:

1. Edit your **/etc/rc_conf** to include the following lines:

```
pf_enable="YES"            # Enable PF (load module if
required)
pflog_enable="YES"         # start pflogd(8)
At this point, you may have to reboot the computer.
```

2. Enable PF with the following command. If PF is already enabled, go to step 3:

```
% sudo pfctl -e
```

3. On FreeBSD and PC-BSD base systems, the **/etc/rc.d/pf** script requires at least a line in **/etc/rc.conf** that reads `pf _ enable="YES"` and a valid **/etc/pf.conf** file. If either of these requirements isn't met, the script will exit with an error message. There is no **/etc/pf.conf** file in a default FreeBSD and PC-BSD installation, so you'll need to create one before you reboot the system with PF enabled. You can create an empty **/etc/pf.conf** with `touch` or could also work from a copy of the **/usr/share/examples/pf/pf.conf** file supplied by the system.

4. With the lines from **step 1** in your **/etc/rc.conf** and created an **/etc/pf.conf** file, you could also use the PF **rc** script to run PF. The following starts the PF daemon:

```
% sudo /etc/rc.d/pf start
```

5. Use the `pfctl` command to modify the behavior of PF—for example, to add or modify packet filter rules.
6. To stop the PF daemon, use the following command:

```
%sudo /etc/rc.d/pf stop
```

PF on Solaris

To run PF as your firewall on Solaris, you first install the PF firewall package, and then configure the firewall to reflect your policy. Once configured, you then enable the firewall service. The PF firewall package can coexist with the IP Filter (ipfilter) package that is preinstalled and enabled by default. However, only one firewall at a time can be enabled on a Solaris system!

To install and configure the PF firewall package, you must become the superuser. The following steps allow you to install, configure, and make the PF firewall the active packet filtering software:

1. Install the PF package with the following command:

```
# pkg install firewall
```

2. Create or update your packet filtering rule set by using the `pfconf` script with the command:

```
# pfconf
```

   The `pfconf` script modifies the **pf.conf** PF configuration file.

   Use the `pfctl` command to modify the behavior of PF—for example, to add or modify packet filter rules.

3. Disable the ipfilter service first, then enable PF. This is critical, because you don't want two active firewall programs with conflicting rules. Use the `svcadm` command to accomplish this as follows:

```
# svcadm disable network/ipfilter
# svcadm enable network/firewall
```

   If the PF configuration file is empty and you enable the firewall service, some traffic filtering occurs. For example, PF drops TCP packets with invalid flag combinations.

4. (Optional) To disable the service, use the `svcadm` command:

```
# svcadm disable network/firewall
```

This command removes all rules from the kernel and disables the service.

The following is an example session that achieves the preceding processes. Note that it does not include modifying the **pf.conf** file:

```
bob@solaris113beta:~$ su
Password: xxx
root@solaris113beta:~# pkg install firewall
           Packages to install:   1
           Services to change:   1
           Create boot environment:  No
           Create backup boot environment: Yes
DOWNLOAD             PKGS         FILES      XFER (MB)    SPEED
Completed            1/1          30/30      0.3/0.3      145k/s
PHASE                                          ITEMS
Installing new actions                         87/87
Updating package state database                 Done
Updating package cache                           0/0
Updating image state                            Done
Creating fast lookup database                   Done
Updating package cache                           1/1
root@solaris113beta:~# svcadm disable network/ipfilter
root@solaris113beta:~# svcadm enable network/firewall
root@solaris113beta:~#
```

23.10 VIRTUALIZATION METHODOLOGIES

A virtual environment for a computer program, and for an operating system, can be defined as a shell within which the program functions autonomously. In Chapter 25 we show three popular and important facilities for creating a virtual operating system environment within a UNIX host environment: PC-BSD Jails implemented and controlled by the iocage program, Solaris 11 Zones, and VirtualBox VMs. These facilities provide extensions of some of the topics we covered in this chapter.

What differentiates these three facilities is that, for the first two, all virtual environments are running under the same kernel on one host machine. In the third one, any number of different kernels can be running simultaneously on one host machine. That means that for Jails you can be running many versions of only PC-BSD at the same time on one machine. And for Zones you can only be running multiple versions of Solaris 11 at the same time on one machine. With VirtualBox, you can be running PC-BSD, Solaris 11, and any number of other operating systems at the same time on one machine.

The important application of these methodologies in the context of system administration is to provide a measure of system security. For example, it is possible with all three facilities to isolate a system service or application program in a guest operating environment, completely autonomous from the host operating system. A service like an ftp server can be run inside of a PC-BSD Jail or Solaris 11 Zone, and anything that intrudes upon that server and its system space does not intrude upon the host operating system space.

If Internet traffic to and from the ftp server is compromised in any way, the server can be stopped and possibly restarted without affecting the host operating system. Another example would be if a faulty, bug-ridden application program were run in a guest environment, it could bring the guest operating system kernel to a halt without in any way affecting the host operating system. Those two example cases are probably the most useful aspects of maintaining a virtual environment, but there are others. We encourage you to go through all of the examples shown in Chapter 25.

SUMMARY

In this chapter, we used a "learning by doing" approach to accomplish the following common system administration tasks:

1. Do a fresh install of a 64-bit version from DVD media using a GUI installer onto a single hard disk system, with a KDE4 or GNOME GUI desktop. Do a preliminary configuration of that system.
2. Illustrate how to gracefully bring the system down.
3. Add additional users to the system and show how to design and maintain user accounts.
4. Adding hardware to the system, particularly disk drives.
5. Provide strategies using the traditional and generic UNIX commands, to backup and archive the system files and user files.
6. Provide a practical method to upgrade and maintain the operating system, and add/update/remove user application package repository software to both increase functionality and update existing packages.
7. Monitor the performance of the system and tune it for optimal performance characteristics.
8. Provide strategies for system security to harden the individual desktop computer.
9. Provide network connectivity strategies, both on a LAN and the Internet.

We showed how to do these common tasks both in PC-BSD and Solaris, the representative systems of the two major families of UNIX.

QUESTIONS AND PROBLEMS

1. Write a brief outline of how you installed your version of the UNIX operating system on your computer. If you didn't do the installation, find out from the system manager how the installation was done and why it was done in that way. If you did a server install, explain how and why you did that.
2. Do the following steps in order to complete the requirements for this problem:

a. Use the `adduser` command to create an anonymous ftp user account on your PC-BSD system with the following configuration:

```
Username: ftp
Full name: Anonymous FTP user
Shell (sh csh tcsh bash rbash zsh nologin) [sh]: nologin
Home directory [/home/ftp]: /var/ftp
Use password-based authentication? [yes]: no
```

Everything else should be left at its default setting.

b. Type the following as superuser to create a new directory and log file for the ftp user:

```
# mkdir -p /var/ftp/pub
# chown ftp:ftp /var/ftp/pub
# touch /var/log/ftpd
```

c. Add a welcome message with your favorite text editor to **/etc/ftpwelcome**

d. With your favorite text editor, create a blank file **/etc/ftpmod**

e. If you have not already done so, enable and start up the ftpd server, as shown in Section 23.2.5.5, Example 23.1. Also add `-S -A -r` options to the **/etc/inetd.conf** file for the ftp service, at the end of the line in that file.

f. Test your anonymous ftp account by using the command `ftp 0` with username **ftp** or **anonymous** and no password. Test it from the Internet. Put files in the user account **/var/ftp**, and retrieve files from that account locally from another account and from the Internet. It is isolated from the rest of the system.

3. Given the 10 steps needed to accomplish user account management shown in Section 23.3, make a table or chart of what users and groups need to be added to your system, and what their default account parameters and group memberships should be. Then, use the Solaris methods shown in Sections 23.3.3.1 through 23.3.3.10 to accomplish user account creation, modification, and deletion from the command line. What command can you use to identify all existing groups on the system? Is there a batch mode option to `useradd` that allows you to create multiple user accounts simultaneously?

4. Add a network printer to your PC-BSD system, and outline the steps necessary to get the printer to actually work given your installation type.

5. If you haven't already done so, do the prerequisites and Examples 16.26 through 16.29 in Chapter 16, Section 16.4.2.1, without using Python. Just use the UNIX commands shown imbedded in the Python code to achieve the same results. These examples illustrate the use of the `rsync` command to do file backup.

6. What is the meaning of the term *archive*?
7. What is the `tar` command used for? Give all its uses.
8. You want to create a tar archive of a project that contains several directories, subdirectories, and files, and save the archive on a USB thumb drive mounted on your system so that you can distribute the archive to your friends.
   a. What is the pathname to a USB thumb drive mounted on your system?
   b. How would you designate a USB thumb drive as the destination for where the tar archive would be created, as an argument to the `tar` command?
9. What are the access permissions for the files on the USB thumb drive from Problem 8?
10. Give a command line for creating a tar archive of your current directory.
11. Give commands for compressing and keeping the archive in the backups directory in your home directory.
12. Give commands for restoring the backup file in Problem 11 in the ~/**backups** directory.
13. Give a command line for copying your home directory to a directory called **home.back** so that access privileges and file modify time are preserved.
14. Why is the `tar` command preferred over the `cp -r` command for creating backup copies of directory hierarchies?
15. Suppose that you download a file, **unixbook.tar.Z** from an ftp site. Give the sequence of commands for restoring this archive and installing it in your ~/**unixbook** directory.
16. Use the `gtar` command to create a compressed archive of a directory of your choosing in a new directory you create named **backups** under your home directory. Name the compressed archive **something.tar.gz**, where **something** is the name of the directory you chose to backup. Show the command lines that you used to perform these tasks.
17. Use the `gtar` command to restore the compressed tar archive ~/**backups/something.tar.gz** you produced in Problem 23.16 into a new directory named **mirrors** under your home directory. Show the command lines that you used to perform these tasks.
18. Use the latest version of Clonezilla “live” to make a bootable clone of your PC-BSD system disk. The source and target disks for cloning can be either both internally mounted, or in the case of a laptop computer, internally mounted for the source, and externally mounted in a USB or SATA enclosure for the target. The instructions for using Clonezilla to do this procedure are found online at the Clonezilla website. Make sure the target disk has a large enough capacity to achieve the cloning!

    To test the clone, gracefully shut down your system and remove the original source system disk. Then replace it with the cloned target and restart the system.

19. Mansoor is working with Bob on a project. He needs to be able to read, write, create, and delete files related to the project, which are located in the **Project** directory in Bob's home directory. Bob and Mansoor are ordinary users without administrative privileges. They wish to do this project without contacting the system administrator to request new groups, group membership changes, `sudo` changes, etc. When the project is over, Bob will remove the modify permissions on his home and the **Project** directory for user **mansoor** himself, instead of contacting the system administrator.

    On your own PC-BSD system, in conjunction with another user, use ACLs to accomplish the following (substituting valid usernames on your system for **bob** and **mansoor**):

    a. Create a project directory under **bob**'s home directory named **Project**.
    b. Set the ACL on **bob**'s home directory so **mansoor** has read, write, and execute privileges on it.
    c. Set the ACL on the **Project** directory so that **mansoor** has `rwxo` privileges on it.
    d. Have **bob** create some files in the **Project** directory.
    e. Have **mansoor** make **bob**'s home directory the current directory.
    f. Have **mansoor** test whether or not he can:
        - Delete files in **bob**'s home directory
        - Delete the **Project** directory from **bob**'s home directory
        - List, create new files, or remove the files that **bob** put in the **Project** directory
    g. Have **bob** revoke **mansoor**'s `x` privileges on **bob**'s home directory and the directory **Project**.
    h. Have **mansoor** test the revocation of modify privileges from step g.

        Show verification of ACL settings at as many steps as necessary to validate what you have done.

20. If you give a set of users permissions to a project directory using ACLs, how can you ensure that subdirectories that are created by the project manager beneath that project directory provide the same access privileges to those users?

21. Create a project directory on your system and create a Git repository in it for any number of local users on your computer system. Then use ACLs to give access to the project directory to the users that are collaborating in the project. This should allow those users to push to and pull from the Git repository. Have your allowed users test the repository. Also, test the security of the repository (i.e., can nonallowed users access it?).

    See Chapter 17, Section 5.7, for more information on creating a Git repository.

22. Adapt the Python code shown in Example 23.7 for your own particular use case.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

CHAPTER 24

ZFS Administration and Use

Objectives

- To describe and give an overview of the Zettabyte File System (ZFS)
- To illustrate the use of the `zpool` and `zfs` commands in the context of system administration
- To give a brief ZFS commands and operations reference encyclopedia
- To give a complete example of file system backups using `zfs snapshot` in a Bourne Shell script
- To detail the Solaris management of ACLs on ZFS files and directories with several examples
- To cover the commands and primitives

  `zpool, zfs, ls -v, chmod`

24.1 INTRODUCTION

This chapter will detail the hands-on mechanics of a modern UNIX file system commonly known as the Zettabyte File System (ZFS). This chapter also assumes that you have already looked at Chapter 23, Section 23.4.3, "Adding a New Disk to the System," particularly the PC-BSD procedures, to gain some exposure to the GUI disk manager and its facilities in PC-BSD.

ZFS has the following attributes: it is self-correcting at the bit level, it is secure, it is a volume manager, and it provides its own file backup system. We show examples of using the two most important ZFS commands, `zpool` and `zfs`, using both Solaris and PC-BSD.

There are very few differences between using those commands in Solaris and PC-BSD. At the time this book was written, the only difference in using ZFS commands between the two systems was in Solaris's use and management of ACLs, the Solaris-only `zfs monitor` command, and pathname specifications for command or subcommand arguments.

24.1.1 zpool and zfs Command Syntax

The following are the general syntax forms for the `zpool` and `zfs` commands.

SYNTAX

```
zpool subcommand [options] [option arguments] [command arguments]
```

Purpose: To create and manage storage pools of virtual devices such as disk drives

Commonly used options/features:

`zpool create name vdev`	Creates a new pool with **name** on the specified **vdev**
`zpool create -o copies=2 name`	Creates a new pool **name** with the property **copies** set to 2
`zpool destroy name`	Destroys, or removes, a pool **name**
`zpool list name`	Lists storage space and health of pool **name**
`zpool scrub name`	Verifies that the checksums on pool **name** are correct
`zpool status name`	Displays the status of pool **name**

SYNTAX

```
zfs subcommand [options] [option arguments] [command arguments]
```

Purpose: To create and manage datasets or file systems mapped to devices such as disk drives

Commonly used options/features:

`zfs create name`	Creates a dataset with **name**
`zfs create -o copies=2 name`	Creates a dataset **name** with the property **copies** set to 2
`zfs destroy name`	Destroys, or removes, a dataset **name**
`zfs list`	Lists all datasets
`zfs rollback name`	Returns dataset **name** to a previous snapshot state

24.1.2 ZFS Terminology

The following describes the basic terminology used throughout this chapter:

Boot environment: A boot environment is a bootable environment consisting of a ZFS root file system and, optionally, other file systems mounted underneath it. Exactly one boot environment can be active at a time.

Checksum: A 256-bit hash of the data in a file system block. The checksum capability can range from the simple and fast fletcher4 (the default) to cryptographically strong hashes such as SHA256.

Clone: A file system whose initial contents are identical to the contents of a snapshot.

Dataset: A generic name for the following ZFS components: clones, file systems, snapshots, and volumes. Each dataset is identified by a unique name in the ZFS name space. Datasets are identified using the following format:

`pool/path[@snapshot]`

`pool`	Identifies the name of the storage pool that contains the dataset
`path`	A slash-delimited pathname for the dataset component
`snapshot`	An optional component that identifies a snapshot of a dataset

Deduplication: Data deduplication is a method of reducing storage capacity needs by eliminating redundant data. Only one unique instance of the data is actually retained on storage media. Redundant data is replaced with a pointer to the unique data copy.

Filesystem: A ZFS dataset of type file system that is mounted within the standard system namespace and behaves like other file systems.

Mirror: A vdev that stores identical copies of data on two or more disks. If any disk in a mirror fails, any other disk in that mirror can provide the same data.

Pool: A logical group of devices describing the layout and physical characteristics of the available storage. Disk space for datasets is allocated from a pool.

RAIDZ: A virtual device that stores data and parity on multiple disks.

Resilvering: The process of copying data from one device to another device is known as resilvering. For example, if a mirror device is replaced or taken offline, the data from an up-to-date mirror device is copied to the newly restored mirror device. This process is referred to as mirror resynchronization in traditional volume management.

Slice: A disk partition created with partitioning software.

Snapshot: A read-only copy of a file system or volume at a given point in time.

Vdev (virtual device): A whole disk, a disk partition, a file, or a collection of the previous, usually all of the same type. On PC-BSD, there is no performance penalty for using disk partitions rather than entire disks. On Solaris, the write cache is disabled for partitions, thus incurring a performance penalty. In both systems, using files as vdevs is discouraged, except for testing purposes. A collection of vdevs is a mirror.

Volume: A dataset that represents a block device. For example, you can create a ZFS volume as a swap device.

24.1.3 How ZFS Works

Create zpool mapped to vdev > Create ZFS file system(s) on zpool > Add files

Simply stated, you create a named zpool first, which at the time it is created is mapped or associated with a vdev, such as a hard disk drive. Then you create one or more file systems in that zpool. Then you add files to the file system(s). Finally, you manage the files, file systems, pools, and vdevs using the appropriate ZFS commands.

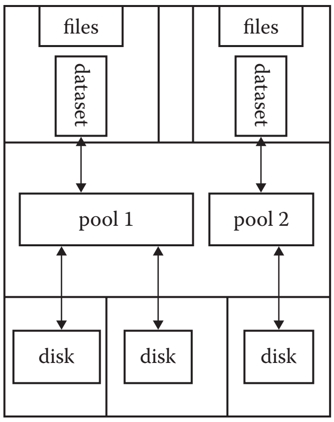


FIGURE 24.1 ZFS component relationships.

Working with ZFS in UNIX is a matter of efficiently and easily managing zpools, vdevs, file systems, and files.

Figure 24.1 shows this relationship between files, datasets (file systems), pools, and disks. **Pool 1** has two disks mapped to it, and a dataset with a number of files in it. **Pool 2** has a single disk mapped to it, and has a dataset in it. This layering of files and datasets, pools, and disks is the basic structure of ZFS.

24.1.4 Important ZFS Concepts

Some very important points have to be made here:

1. Only one zpool can be mapped or associated with any vdev. So if you want to create a zpool on a physical hard disk or one of its slices, no other existing active zpool can be mapped to that vdev!
2. There are seven types of vdev in ZFS:
    - Disk (default): The physical hard drives in your system, usually the whole drive or primary slice
    - File: The absolute path of preallocated files/images, similar to the Section 24.2.2, Example 24.1
    - Mirror: Standard software RAID1 mirror
    - Raidz1/2/3: Nonstandard distributed parity-based software RAID levels
    - Spare: Hard drives marked as a *hot spare* for ZFS software RAID
    - Cache: Device used for a level-2 adaptive read cache (L2ARC)
    - Log: A separate log (SLOG) called the ZFS intent log (ZIL)

3. Unlike a traditional file system, where the mount point of the file system begins at a particular logical drive letter, the default mount point for a zpool is root (/).

   This is how the path to a file named **test.txt** appears when it is in the zpool named **data1** on the file system **sarwar**:

   ```
   /data1/sarwar/test.txt
   ```

   Here's how the path to a file named **test.txt** appears on a traditional file system:

   ```
   C:\Users\Robert\Desktop\test.txt
   ```

   When you want a ZFS file system to expand onto more than one disk, for example, you add more disks to the zpool.

4. A zpool can be enlarged by adding more devices, but it cannot be shrunk (at least not at this time)!

24.2 EXAMPLE ZFS POOLS AND FILE SYSTEMS: USING THE `ZPOOL` AND `ZFS` COMMANDS

In this section, we first reiterate a set of simple methods given in Chapter 23, to allow you to quickly determine the logical device names of disks attached to your UNIX system.

We then present five examples that will give you some basic experience in using the `zpool` and `zfs` commands.

24.2.1 A Quick and Easy Way to Find Out the Logical Device Names of Disks Actually Installed on Your System

These techniques, which were presented in Chapter 23, are worth repeating here in preparation for using the `zpool` and `zfs` commands.

It is important to know how to determine, in a very quick and easy manner, what the currently installed logical device names of disk drives actually attached and usable on your system are. What we mean by "attached and useable" is that the disk drive is properly connected and recognized by the system and has a device driver that the system can use to communicate with it.

The simple methods that follow show how to determine what disk drives are attached and usable on your system, and what the logical device names of those and any others you might want to add to your system are. These methods are done differently in Solaris and PC-BSD:

Solaris methods: For SCSI, SATA or IDE bus hard disk drives, type `format` on the command line as superuser. The output of this command shows your main hard drive, for example, disk **0** as **/dev/dsk/c0d0s0**. Type `0` to see the specifications of disk **0**. When the `format>` prompt appears, type `quit` to leave the `format` facility.

For USB thumb drives or hard disk drives, type `rmformat` on the command line as superuser. The output of this command shows, for example, a USB thumb drive as **/dev/**

rdsk/c9t0d0p1. A CD or DVD writer would show up here too, usually not on the USB bus, but on the SATA bus.

Additionally, you can use the Gparted Partition Editor as shown in Chapter 23, Example 23.3.

PC-BSD methods: Change your current working directory to **/dev**. Type `ls`. Hard drives, for example, show up in the `ls` listing as **ada0**, **ada1**, and so on. The full path to the first slice on one of these disks is specified as **/dev/ada0s1**. A USB device, like a thumb drive, would show up in the `ls` listing as **da0**, and the full path to the first slice on it would be **/dev/da0s1** or **/dev/da0p1**.

Additionally, you can type `gpart show` on the command line in a terminal window, and it will list the drives available for slicing, such as **ada0**, **ada1s1**, **da0s1**, etc., for disks and USB devices. The logical device names to all of those devices is provided in a compact and easily understood listing.

24.2.2 Basic ZFS Examples

In this section we present six instructive, introductory examples of how to work with ZFS. It is expected that for you to get the full benefit from them, you do them and their attendant in-chapter exercises in the order presented.

The examples are as follows:

Example 24.1 The `zpool` Command: Using Files Instead of Disks as Vdevs

Example 24.2 The `zfs` Command: `send`, `receive`, `snapshot`

Example 24.3 Mirroring of Hard Disks on Solaris

Example 24.4 Mirroring of Hard Disks on PC-BSD

Example 24.5 Creating a ZFS Pool on a USB Thumb Drive for Solaris

Example 25.6 Creating a ZFS Pool and Dataset on a USB Thumb Drive on PC-BSD

Example 24.1: The `zpool` Command: Using Files Instead of Disks as Vdevs

Objective: To introduce the `zpool` command, implemented on files instead of disks, and to show forms of ZFS pool creation and mirroring.

Introduction: A vdev, as defined previously, can be a physical device such as a disk drive, a file, a single slice on a hard disk drive, or a collection of devices. Before beginning to use ZFS on physical devices, and to practice using ZFS on an existing file system instead of deploying ZFS on actual SATA hard disk drives, we will create and manipulate files with the important ZFS commands.

Also, if you do not have a second hard disk drive in your computer, you can do this example to gain an appreciation of what ZFS is.

The only difference between Solaris and PC-BSD in this example is the use of the `mkfile` command. The following commands that use `mkfile` to create empty files quickly are for Solaris. The PC-BSD equivalent of the `mkfile` command is the `truncate` command. In PC-BSD, substitute the command `truncate -s 128m filename` for the following `mkfile` command lines.

In case you want to use four real disks mounted and partitioned in this preliminary introductory example, make a note of the full path to their device names (e.g., **/dev/dsk/c2t1d0p1** under Solaris, or **/dev/ada0s1** under PC-BSD). You will be destroying all the partition information and data on these disks, so be sure they're not needed!

If you make a mistake anywhere along the way, you can always start over by executing the cleanup steps shown at the end of the example and begin again.

Prerequisites: An installation of either Solaris or PC-BSD on your system.

Procedure: Follow the steps in the order shown to complete this example.

1. To begin, create four 128 MB files as follows (the files must be a minimum of 64 MB in size):

```
root@solaris:~# mkfile 128m /home/bob/disk1
root@solaris:~# mkfile 128m /home/bob/disk2
root@solaris:~# mkfile 128m /home/bob/disk3
root@solaris:~# mkfile 128m /home/bob/disk4
root@solaris:~# ls -lh /home/bob
total 1049152
-rw------T   1 root     root        128M Mar  7 16:48 disk1
-rw------T   1 root     root        128M Mar  7 16:48 disk2
-rw------T   1 root     root        128M Mar  7 16:48 disk3
-rw------T   1 root     root        128M Mar  7 16:48 disk4
```

   In this example, we initially create and use files to simulate disks on an already existing file system, and we named them **disk1**, **disk2**, **disk3**, and **disk4** to enhance that illusion.

   Also, it is assumed in the example code that the current working directory is **/home/bob** unless otherwise noted.

2. Before creating new pools you should check for existing pools to avoid confusing them with the example pools we create here. You can check what pools exist with `zpool list`:

```
root@solaris:~# zpool list
```

3. Pools are created using `zpool create`. We can create a single disk pool using a file as follows (you must use the absolute path to the file):

```
root@solaris:~# zpool create data /home/bob/disk1
root@solaris:~# zpool list
NAME    SIZE   USED   AVAIL   CAP  DEDUP  HEALTH   ALTROOT
The default pools on the system...
data    123M   51.5K   123M   0%   1.00X  ONLINE    -
```

4. Now we will create an actual file in the new pool:

```
root@solaris:~# mkfile 32m /data/data20file
root@solaris:~# ls -lh /data/data20file
-rw------T   1 root    root     32M Mar  7 16:56 /data/
data20file
root@solaris:~# zpool list
NAME    SIZE   USED   AVAIL   CAP  DEDUP  HEALTH   ALTROOT
The default pools on the system...
data    123M   32.1M  90.9M   26%  1.00X  ONLINE    -
```

5. We will now destroy the pool data with `zpool destroy`:

```
root@solaris:~# zpool destroy data
root@solaris:~# zpool list
```

Only the default pools on the system are shown.

6. Creating a Mirrored Pool with Files
   A pool composed of a single disk doesn't offer any redundancy! One way of providing protection against physical disk failure is to use a mirrored pair of disks in a pool:

```
root@solaris:~# zpool create data2 mirror /home/bob/disk1
/home/bob/disk2
root@solaris:~# zpool list
NAME    SIZE   USED   AVAIL   CAP  DEDUP  HEALTH   ALTROOT
The default pools on the system...
data2   123M   51.5K  123M    0%   1.00X  ONLINE    -
```

7. To get more information about the pool **data2**, we use `zpool status`:

```
root@solaris:~# zpool status data2
pool: data2
state: ONLINE
scrub: none requested
config:
    NAME              STATE   READ WRITE CKSUM
    data2               ONLINE    0    0    0
     mirror              ONLINE    0    0    0
      /home/bob/disk1    ONLINE    0    0    0
      /home/bob/disk2    ONLINE    0    0    0
errors: No known data errors
```

8. Now create a file in the **data2** pool.

```
root@solaris:~# mkfile 32m /data2/data2file
```

Note the change in the pool after we have added a file to it.

```
root@solaris:~# zpool list
NAME     SIZE   USED  AVAIL   CAP  DEDUP  HEALTH   ALTROOT
```

```
The default pools on the system...
data2    123M  32.1M  90.9M    26%  1.00X  ONLINE    -
```

About a quarter of the disk has been used, but more importantly the data is now stored redundantly over two disks.

9. Let's test that redundancy by overwriting the first "disk" label with random data. If you are using real hard disks, you could physically remove the disk from the computer.

```
root@solaris:~# dd if=/dev/random of=/home/bob/disk1
bs=512 count=1
1+0 records in
1+0 records out
```

10. ZFS automatically checks for errors when it reads/writes files, but we can force a check with the `zfs scrub` command.

```
root@solaris:~# zpool scrub data2
```

11. Let's check the status of the pool:

```
root@solaris:~# zpool status
pool: data2
state: DEGRADED
status: One or more devices could not be used because the
label is missing or invalid. Sufficient replicas exist for
the pool to continue
  functioning in a degraded state.
action:    Determine if the device needs to be replaced,
and clear the errors using zpool clear or fmadm repaired,
or replace the device using 'zpool replace'.
       Run zpool status -v to see device specific details.
scan: scrub completed with 0 errors on Wed Mar 6 17:42:07 2014
config:
  NAME              STATE     READ WRITE CKSUM
  Data2             DEGRADED  0     0     0
   mirror -0        DEGRADED  0     0     0
   /home/bob/disk1  UNAVAIL   0     0     0 corrupted data
   /home/bob/disk2  ONLINE    0     0     0

errors: No known data errors
```

12. The disk we used `dd` on is showing as `UNAVAIL` (unavailable) with corrupted data, but no data errors are reported for the pool as a whole, and we can still read and write to the pool:

```
root@solaris:~# mkfile 32m /data2/data2file2
root@solaris:~# ls -l /data2/
total 131112
-rw------T  1 root   root  33554432 Mar 6 17:43 data2file
-rw------T  1 root   root  33554432 Mar 6 17:35 data2file2
```

13. To maintain redundancy we should replace the broken disk with another. If you are using a physical disk you can use the `zpool replace` command (the zpool man page has details). However, in this file-based example we will just remove the disk file from the mirror and recreate it.

    Devices are detached with `zpool detach`:

```
root@solaris:~# zpool detach data2 /home/bob/disk1
```

14. Let's check the status of the pool:

```
root@solaris:~# zpool status data2
pool: data2
state: ONLINE
scrub: scrub completed with 0 errors on Thurs Mar 6
18:00:07 2014
config:
    NAME                    STATE      READ WRITE CKSUM
    data2                   ONLINE       0     0     0
     /home/bob/disk2        ONLINE       0     0     0
errors: No known data errors
```

15. Let's remove the disk to simulate a failure and then replace it:

```
root@solaris:~# rm /home/bob/disk1
root@solaris:~# mkfile 128m /home/bob/disk1
```

16. In order to replace it in the mirror, we need to do the following. To attach another device we specify an existing device in the mirror to attach it to with `zpool attach`:

```
root@solaris:~# zpool attach data2 /home/bob/disk2 /home/
                bob/disk1
```

17. Check the status of the pool:

```
root@solaris:~# zpool status data2
pool: data2
state: ONLINE
scrub: resilver completed with 0 errors on Thurs Mar 6
18:08:16 2014
config:
    NAME                    STATE      READ WRITE CKSUM
    data2                   ONLINE       0     0     0
     mirror  -0             ONLINE       0     0     0
      /home/bob/disk2       ONLINE       0     0     0
      /home/bob/disk1       ONLINE       0     0     0
errors: No known data errors
```

18. Adding to a Mirrored Pool

    A very critical systems administration procedure accomplished by ZFS is to add disks to a pool without taking it offline. Let's double the size of our **data2** pool:

```
root@solaris:~# zpool list
NAME    SIZE   USED   AVAIL   CAP  DEDUP  HEALTH   ALTROOT
The default pools on the system...
data2      123M   64.5M  58.5M   52%  1.00X  ONLINE    -
```

19. We can use the `zpool add` command to add disks to the existing pool.

```
root@solaris:~# zpool add data2 mirror /home/bob/disk3 /
                home/bob/disk4
root@solaris:~# zpool list
NAME SIZE USED AVAIL CAP DEDUP HEALTH ALTROOT
The default pools on the system...
data2      246M   64.5M  181M    26%   1.00X  ONLINE    -
```

20. The file systems within the pool are always available. If we look at the status now, it shows the pool consists of two mirrors:

```
root@solaris:~# zpool status data2
pool: data2
state: ONLINE
scrub: resilver completed with 0 errors on Wed Mar 6
17:58:17 2014
config:
    NAME                  STATE     READ WRITE CKSUM
    data2                 ONLINE       0     0     0
     mirror  -0           ONLINE       0     0     0
      /home/bob/disk2     ONLINE       0     0     0
      /home/bob/disk1     ONLINE       0     0     0
     mirror  -1           ONLINE       0     0     0
      /home/bob/disk3     ONLINE       0     0     0
      /home/bob/disk4     ONLINE       0     0     0

errors: No known data errors
```

21. We can see where the data is currently written in our pool using `zpool iostat -v`:

```
root@solaris:~# zpool iostat -v data2
                   capacity      operations    bandwidth
pool              alloc  free     read  write   read  write
------------      -----  -----   -----  -----   -----  -----
data2             64.5M  181M       0      0    14.7K  623
 mirror           64.4M  58.6M      0      0    40.5K 1.51K
  /home/bob/disk2   -       -       0      2    22.4K 22.8K
  /home/bob/disk1   -       -       0      1    187   43.6K
 mirror            102K  123M       0      0    0     2.32K
  /home/bob/disk3    -     -        0      0    213   22.9K
  /home/bob/disk4    -     -        0      0    213   22.9K
----------------   -----  -----    -----  -----   -----  -----
```

22. All the data is currently written on the first mirror pair and none on the second. This makes sense, as the second pair of disks was added after the data was written. If we write some new data to the pool, the new mirror will be used:

```
root@solaris:~# mkfile 64m /data2/data2file3
root@solaris:~# zpool iostat -v data2
                    capacity      operations    bandwidth
pool               alloc  free    read  write   read  write
------------       -----  -----   -----  -----  -----  -----
data2              129M   117M      0      1     14.4K 17.2K
 mirror            83.6M  39.4M     0      1     38.5K 15.4K
  /home/bob/disk2    -      -       0      2     22.0K 27.6K
  /home/bob/disk1    -      -       0      1     177   55.3K
 mirror            45.4K  77.6M     0      0     0     256K
  /home/bob/disk3    -      -       0      9     121   268K
  /home/bob/disk4    -      -       0      9     121   268K
------------------ -----  -----   -----  -----  -----  -----
```

23. We see how a little more of the data has been written to the new mirror than to the old: ZFS tries to make the best use of all the resources in the pool. Now do these in-chapter exercises, and then continue onto the next step.

EXERCISE 24.1

If you have not already done so, execute all of the steps of Example 24.1 in PC-BSD using proper commands and pathnames.

EXERCISE 24.2

In Example 24.1, step 4, what is the pathname to **datafile20**?

EXERCISE 24.3

If you were to use a text editor like emacs to create a text file named **text1.txt** in the file system named **data**, how would you designate the complete pathname to that text file?

EXERCISE 24.4

In Example 24.1, after step 6 was executed correctly, and you created a text file with emacs in the **data2** file system, would the pathnames to the two mirrored versions of that text file be different? In other words, could you edit each one of them separately by designating different pathnames to them?

EXERCISE 24.5

In Example 24.1, step 19, could you add a single disk into the mirrored **data2** zpool, instead of the two disks specified?

EXERCISE 24.6

In Example 24.1, step 20, are the mirrors named **mirror-0** and **mirror-1** mirrors of each other?

24. To clean up after doing our work, let's delete everything we created in this example.

    From the root directory, destroy the **data2** file system and its files.

```
root@solaris:~# zfs destroy -r data2
```

25. Next, destroy the **data2** zpool.

```
root@solaris:~# zpool destroy data2
```

26. Finally, destroy the disk simulation files.

```
root@solaris:~# rm /home/bob/disk*
root@solaris:~#
```

Conclusion: We can use the `zpool` command and its `create` subcommand to associate or map file systems to vdevs, whether the vdev is a file itself or a disk drive.

Example 24.2: The `zfs` Command: `Send` and `Receive`, `Snapshot`

Objectives: The following is a complete Solaris example of using the commands `zfs send` and `zfs  receive`. Its objective is to show how to create a file system with `zfs` and work with file systems.

Introduction: In the example, we backup a file system with an incremental update, from one file system to another, on the same zpool and vdev. With just the minor syntactic changes shown, it can be adapted to PC-BSD.

As with Example 24.1, this example creates a file in your home directory that *emulates* a vdev, so you don't have to have a second hard disk available! This is the easiest, most cost-effective technique, and the best for practicing and developing your basic skills with ZFS.

What the example does: It backs up a file system named **data** in the zpool named **sender** to another file system named **backup** in the same pool. The data file system contains a file we create named **test.txt**. It uses the `snapshot` subcommand of the `zfs` command to achieve this.

If you make a mistake anywhere along the way, you can always start over by executing the cleanup steps shown at the end of the example and begin again.

Prerequisites: An installation of either Solaris or PC-BSD on your system, having complete Example 24.1.

Procedure: To accomplish the objectives of this example, do the following steps in the order presented.

1. Become the superuser.

```
bob@solaris:~$ su
```

2. Create the vdev as a file.

```
root@solaris:~# mkfile -v 100m /home/bob/master
In PC-BSD use this command-
# truncate -s 100m /usr/home/bob/master
```

3. Create a zpool in that vdev named **sender**.

```
root@solaris:~# zpool create sender /home/bob/master
In PC-BSD use this command:
# zpool create sender /usr/home/bob/master
```

4. Create a ZFS file system, named **data**, in the **sender** zpool.

```
root@solaris:~# zfs create sender/data
```

5. Create a test file in the **sender/data** ZFS file system.

```
root@solaris:~# echo "created: 09:58" > /sender/data/
   test.txt
```

6. Create a snapshot of the ZFS file system named **sender/data**.

```
root@solaris:~# zfs snapshot sender/data@1
```

7. Let's examine the location where the snapshot has been saved. First, use the `zfs list` command with the `snapshot` command argument.

```
root@solaris:~# zfs list -t snapshot
NAME                    USED  AVAIL  REFER  MOUNTPOINT
...<Output truncated>...
sender/data@1             0     -    31.5K      -
```

8. By default the snapshot location is hidden. To unhide it, use the `zfs set` command.

```
root@solaris:~# zfs set snapdir=visible sender/data
```

9. Let's see what the contents of the data file system are.

```
root@solaris:~# ls -la /sender/data
total 8
drwxr-xr-x   3 root     root        3 Oct 25 07:30 .
drwxr-xr-x   3 root     root        3 Oct 25 07:28 ..
dr-xr-xr-x   4 root     root        4 Oct 25 07:28 .zfs
-rw-r--r--   1 root     root       15 Oct 25 07:30 test.txt
```

10. The snapshot directory that contains the first snapshot is under **.zfs**, as shown. So let's change to the directory that contains it, and use `ls –la` to see what is in that directory.

```
root@solaris:~# cd /sender/data/.zfs/snapshot/1
root@solaris:/sender/data/.zfs/snapshot/1# ls -la
total 5
drwxr-xr-x   2 root     root        3 Oct 25 07:30 .
dr-xr-xr-x   3 root     root        3 Oct 25 07:30 ..
-rw-r--r--   1 root     root       15 Oct 25 07:30 test.txt
```

The file **test.txt** in this directory is a “frozen” picture of what was contained in the **/sender/data** file system at the time we did step 6.

11. Return to the root directory.

```
root@solaris:/sender/data/.zfs/snapshot/1# cd
root@solaris:~#
```

12. Create a ZFS file system named **backup** in the **sender** zpool.

```
root@solaris:~# zfs create sender/backup
```

13. Send the snapshot to the backup file system.

```
root@solaris:~# zfs send sender/data@1 | zfs receive -F
                sender/backup
```

After this command executes, the file **test.txt** is in the backup file system.

14. Set the sender/backup file system to read only to prevent data corruption. Make sure to do this before accessing anything in the **sender/backup** file system.

```
root@solaris:~# zfs set readonly=on sender/backup
```

15. Now we will make some changes in the original file. Use `echo` to update the original **test.txt** file to simulate changes in the data file system.

```
root@solaris:~# echo "'date'" >> /sender/data/test.txt
```

16. Create a second snapshot of **sender/data**.

```
root@solaris:~# zfs snapshot sender/data@2
```

17. Send the differences. You may get an error message saying that the destination has been modified if you did not set the **sender/data** file system to read only three commands previously in step 14.

```
root@solaris:~# zfs send -i sender/data@1 sender/data@2 |
zfs receive sender/backup
```

18. Optionally, at this point you could use ssh to send the file system to another zpool on another machine such as **backup_server** (where you need to supply the IP address and go into the root account), as follows:

```
root@solaris:~# zfs send sender/data@1 | ssh backup_server
zfs receive backup/data@1
```

19. Now let’s take a look at what is in the second snapshot directory.

```
root@solaris:~# cd /sender/data/.zfs/snapshot/2
root@solaris:/sender/data/.zfs/snapshot/2# ls -la
total 5
drwxr-xr-x   2 root     root      3  Oct 25 07:30 .
dr-xr-xr-x   4 root     root      4  Oct 25 07:46 ..
-rw-r--r--   1 root     root      58 Oct 25 07:45 test.txt
```

20. Let's look at the contents of the **test.txt** file.

```
root@solaris:/sender/data/.zfs/snapshot/2# more test.txt
created: 09:58
Saturday, October 25, 2014 07:45:16 AM PDT
```

21. Now let's compare what is in the second snapshot directory to the **sender** and **backup** file systems.

```
root@solaris:/sender/data/.zfs/snapshot/2# cd
root@solaris:~# cd /sender/data
root@solaris:/sender/data# ls
test.txt
root@solaris:/sender/data# more test.txt
created: 09:58
Saturday, October 25, 2014 07:45:16 AM PDT
root@solaris:/sender/data# cd ..
root@solaris:/sender# cd backup
root@solaris:/sender/backup# ls
test.txt
root@solaris:/sender/backup# more test.txt
created: 09:58
Saturday, October 25, 2014 07:45:16 AM PDT
```

22. Return to the root.

```
root@solaris:/sender/backup# cd
root@solaris:~#
```

Now do these in-chapter exercises, and then continue onto the next step.

EXERCISE 24.7

If you have not already done so, execute all the steps of Example 24.2 in PC-BSD using proper commands and pathnames.

EXERCISE 24.8

What commands would you use to make the current working directory the one that contains the second snapshot?

EXERCISE 24.9

What are the contents of the first snapshot file, **test.txt**?

EXERCISE 24.10

Redo Example 24.2 using two different zpools named **source** and **target**. Create a file system on a pool source named **origin**, and create a file system on a pool target named **destination**. Instead of using the `echo` command to create the file **test.txt** in the **source/origin** file system, use your favorite text editor, like emacs. Then create a couple of snapshots of **origin** and **destination**, making some changes in **test.txt** with emacs. Finally, use the techniques shown in Example 24.2 to verify that the snapshots indeed contain the changes you made with emacs in **test.txt**.

EXERCISE 24.11

If you have two or more hard disk drives on your UNIX system, redo Exercise 24.8 so that the zpools are created on actual disks rather than on files that simulate disks.

23. To clean up after doing our work, let's delete everything we created in this example. Notice that destroying the datasets destroys the snapshots.
    From the root directory, destroy the **backup** file system and its data.

    ```
    root@solaris:~# zfs destroy -r sender/backup
    ```

24. Next, destroy the **data** file system and its data.

    ```
    root@solaris:~# zfs destroy -r sender/data
    ```

25. Next, destroy the **sender** zpool.

    ```
    root@solaris:~# zpool destroy sender
    ```

26. Finally, delete the disk simulation file.

    ```
    root@solaris:~# rm /home/bob/master
    root@solaris:~#
    ```

    For PC-BSD systems, the syntax for deleting the disk simulation file is:

    ```
    # rm /usr/home/bob/master
    ```

Conclusion: You can use `zfs send/receive` as a backup mechanism, either locally between two hard disks attached to the system, or between systems over a network.

Example 24.3: Mirroring of Hard Disks on Solaris

Objectives: To mirror the *boot disk*, sometimes called the *system disk*, onto another hard disk that is added to the system sometime after the initial build of Solaris.

Introduction: The following example illustrates one of the most important disk maintenance procedures a user can perform: the mirroring of a physical device using `zpool attach`. In the example, we mirror a Solaris boot disk onto another hard disk of equal size. This is important because if one of the hard disks, perhaps your original system disk, fails, you have an exact, bootable duplicate of it in your machine with which you can start up the system again.

The resilvering operation for the 1 TB hard disk in the example, with very little data on it, takes about five to ten minutes.

If you make a mistake anywhere along the way, you can always start over by executing the cleanup step shown at the end of the example and begin again.

Prerequisites:

1. Using Solaris.
2. That you have previously completed Example 24.2.
3. That you have correctly connected and put a single primary partition on a second hard disk on your system.
4. That you have previously determined the logical device name and the full path to it using the methods "A Quick and Easy Way to Find Out the Logical Device Names of Disks Actually Installed on Your System" from Section 24.2.1. The complete logical device names of our main hard disk (the boot disk with the operating system and all your files on it) and the second hard disk drive we want to mirror it to are **/dev/dsk/c7t1d0** and **/dev/dsk/c7t2d0**. On your system they may not be exactly the same, but they will be very similar.
5. The size in bytes of **rpool** (the name of the default system pool for Solaris) is smaller than or equal to the size of the primary partition on the second hard disk drive you will mirror **rpool** to.

Procedure: Do the following steps in the order shown to meet the objectives.

1. Become the superuser.

```
bob@solaris:~$ su
Password: xxx
root@solaris:~#
```

2. Use the `attach` subcommand of zpool to create a mirror of your main system disk. Be sure to specify the complete pathname to the devices, as shown.

```
root@solaris:~# zpool attach rpool /dev/dsk/c7t1d0 /dev/
                dsk/c7t2d0
```

   Make sure to wait until resilver is done before rebooting.

3. While the resilvering is happening, check the status of the resilvering process.

```
root@solaris:~# zpool status
pool: rpool
state: DEGRADED
status: One or more devices is currently being resilvered.
        The pool will continue to function in a degraded
        state.
action: Wait for the resilver to complete.
Run 'zpool status -v' to see device specific details.
scan: resilver in progress since Mon Oct 27 06:30:26 2014
32.4G scanned
9.98G resilvered at 102M/s, 30.80% done, 0h3m to go
config:
      NAME        STATE     READ WRITE CKSUM
      rpool       DEGRADED    0     0     0
        mirror-0  DEGRADED    0     0     0
```

```
          c7t1d0  ONLINE     0     0     0
          c7t2d0  DEGRADED   0     0     0 (resilvering)
```

4. Check the status of the resilvering again. This time, it is done and **rpool** is back online as a two-disk mirror.

```
root@solaris:~# zpool status
pool: rpool
state: ONLINE
scan: resilvered 32.4G in 0h9m with 0 errors on Mon Oct 27
06:40:15 2014
config:
      NAME         STATE     READ WRITE CKSUM
      rpool        ONLINE      0     0     0
        mirror-0   ONLINE      0     0     0
          c7t1d0   ONLINE      0     0     0
          c7t2d0   ONLINE      0     0     0

errors: No known data errors

root@solaris:~#
```

5. Use the `format` command, and its `verify` subcommand, to check that the new disk has been formatted, sliced, and contains a slice **0** for the boot partition information. In the following format display, the original system disk is disk **0** and the newly added disk is disk **1**.

```
root@solaris:~# format
Searching for disks...done
AVAILABLE DISK SELECTIONS:
       0. c7t1d0 <ATA-ST1000DM003-1CH1-CC49-931.51GB>
          /pci@0,0/pci103c,1609@11/disk@1,0
       1. c7t2d0 <ATA-ST1000DM003-1CH1-CC49-931.51GB>
          /pci@0,0/pci103c,1609@11/disk@2,0
Specify disk (enter its number): 1
selecting c7t2d0
[disk formatted]
/dev/dsk/c7t2d0s1 is part of active ZFS pool rpool. Please
see zpool(1M).
FORMAT MENU:
        disk       - select a disk
        type       - select (define) a disk type
        partition  - select (define) a partition table
        current    - describe the current disk
        format     - format and analyze the disk
        fdisk      - run the fdisk program
        repair     - repair a defective sector
        label      - write label to the disk
        analyze    - surface analysis
        defect     - defect list management
        backup      - search for backup labels
        verify      - read and display labels
        inquiry     - show disk ID
        volname     - set 8-character volume name
        !<cmd>      - execute <cmd>, then return
        quit
format> verify

Volume name = <        >
ascii name = <ATA-ST1000DM003-1CH1-CC49-931.51GB>
bytes/sector = 512
sectors = 1953525167
accessible sectors = 1953525134
Part  Tag         Flag  First Sector  Size      Last Sector
  0   BIOS_boot   wm    256           256.00MB  524543
  1   usr         wm    524544        931.26GB  1953508750
  2   unassigned  wm    0             0         0
  3   unassigned  wm    0             0         0
  4   unassigned  wm    0             0         0
  5   unassigned  wm    0             0         0
  6   unassigned  wm    0             0         0
  8   reserved    wm    1953508751    8.00MB    1953525134

format>quit
root@solaris:~#
```

6. We can see from the preceding format verify that the new disk has been formatted, sliced, and contains a slice **0** for the boot partition information. Now we need to make the new disk bootable.

```
root@solaris:~# /usr/sbin/bootadm install-bootloader
root@solaris:~#
```

EXERCISE 24.12

To test the usability of the new hard disk in the mirrored pair, shut down your machine gracefully. Then disconnect and remove the first boot disk from the machine. Finally, reboot the machine with only the new hard disk in the system. What is the status of the pool you attached the second hard disk to as a mirror after a successful reboot? After doing this exercise, you may replace the original boot disk and boot into it normally. Do the remaining step at your discretion.

7. If you want to retain this two-disk mirror, stop. If you want to detach the second hard disk from the pool, thus destroying the mirror, do the following:

```
root@solaris:~# zpool detach rpool /dev/dsk/c7t1d0
root@solaris:~#
```

Conclusion: You have created a two-disk mirror of the boot disk with another equal-sized hard disk using `zpool attach`. Additionally, you have used the `format` command subcommands to check on the integrity of the new disk you added.

Example 24.4: Mirroring of Hard Disks on PC-BSD

Objectives: To mirror the *boot disk*, sometimes called the *system disk*, onto another hard disk that is added to the system sometime after the initial build of PC-BSD.

Introduction: The following example illustrates one of the most important disk maintenance procedures a user can perform: the mirroring of a physical device using `zpool attach`. In the example, we mirror a PC-BSD boot disk onto another hard disk of equal (or greater) size.

This is important because if one of the hard disks, perhaps your original system disk, fails, you have an exact, bootable duplicate of it in your machine with which you can start up the system again. This system disk may also have all of your user datasets on it. You can then replace the failed disk and in a few simple ZFS steps, restoring the integrity and redundancy of your system without taking it offline!

This example accomplishes the same thing that the Clonezilla Live program illustrated in Chapter 23 does, but with one significant difference. Whereas Clonezilla Live "clones" an entire disk, including the boot sectors, at only one discreet instant in time, ZFS `zpool attach` applied to a mirrored pair that includes the system disk creates a constantly mirrored "clone" of the system disk and any other datasets on it. This is critical, because you never know when your system disk is going to fail. You can have any number of backup schemes in place, as shown in Chapter 23 and this chapter as well, to save user datasets with rolling, incremental backups using `rsync` or `zfs snapshot`. But this example's methodology allows you to constantly have an exact clone of your entire system available as long as it is running and active.

Of course, a more advanced and necessarily complex technique for doing what is shown here would involve multiple disks, including a separate system disk, higher-level RAIDZ dataset disks, and even the disk that holds the ZIL.

If you make a mistake anywhere along the way, you can always start over by executing the cleanup step shown at the end of the example and begin again. Depending on how far you go in the procedure, you can also reformat the new disk with the `gpart` command and restart from the beginning.

Prerequisites:

1. Using PC-BSD.
2. That you have previously completed Example 24.3.
3. That you have correctly connected and put a second hard disk onto your system, and it is unpartitioned but perhaps has a GPT partition scheme and table on it. It should also be large enough to accommodate everything on the original system disk partition we are going to use.
4. That you have previously determined the logical device name and the full path to it using the methods "A Quick and Easy Way to Find Out the Logical Device

Names of Disks Actually Installed on Your System" from Section 24.2.1. The complete logical device names, including the appropriate partitions on those disks, of our original hard disk (the boot disk with the operating system and all your files on it) and the second hard disk drive we want to mirror it to are **/dev/ada0s1** and **/dev/ada1p3**. On your system they may not be exactly the same, but they will be very similar.

5. The size in bytes of **tank** (the name of the default system pool for our PC-BSD) is smaller than or equal to the size of the partition on the second hard disk drive you will mirror **tank** to.

Procedure: Do the following steps in the order shown to meet the objectives.

1. Become the superuser.

```
[bob@pcbsd-5867] ~# su
Password: xxx
[bob@pcbsd-5867] ~#
```

2. Having previously determined the logical device name to the new hard disk to be **ada1**, you should delete the ZFS metadata associated with all partitions that may have previously existed on it. This also clears the partitions on **ada1**.

```
[bob@pcbsd-5867] ~# zpool labelclear -f ada1
```

In the next steps, we will prepare partitions on the new hard disk to accommodate the mirroring process. If the new disk does not have a GPT partition table on it, you can use the following `gpart` command:

```
bob@pcbsd-5867]~# gpart create -s gpt ada1
```

3. You then have to add a boot partition to the new hard disk, using the `gpart` command, so that you will be able to boot to it in the event of a failure of the original system disk.

```
[bob@pcbsd-5867] ~# gpart add -b 40 -l gptboot -s 512k -t
                    freebsd-boot ada1
ada1p1 added
```

4. You then have to add a swap space partition to the new hard disk.

```
[bob@pcbsd-5867] ~# gpart add -s 1G -l swap1 -t freebsd-
swap ada1
ada1p2 added
```

5. You then have to add the primary partition that will contain the **tank1** zpool and all file systems attached to it.

```
[bob@pcbsd-5867] ~# gpart add -t freebsd-zfs -l zfs1 ada1
ada1p3 added
```

6. You then have to put the bootloader code on the boot partition you created in step 3.

```
[bob@pcbsd-5867] ~# gpart bootcode -b /boot/pmbr -p /boot/
gptzfsboot -i 1 ada1
bootcode written to ada1
```

7. You can then examine the partition scheme for all disks attached to your system.

```
[bob@pcbsd-5867] ~# gpart show
=>          63    156301425  ada0   MBR   (75G)
            63           63     - free -   (32K)
           126    156301299     1  freebsd   [active]   (75G)
     156301425           63     - free -   (32K)

=>           0    156301299  ada0s1   BSD   (75G)
             0    152086528     1  freebsd-zfs   (73G)
     152086528      4194304     2  freebsd-swap   (2.0G)
     156280832        20467     - free -   (10M)

=>          34    156249933  ada1   GPT   (75G)
            34            6     - free -   (3.0K)
            40         1024     1  freebsd-boot   (512K)
          1064      2097152     2  freebsd-swap   (1.0G)
       2098216    154151751     3  freebsd-zfs   (74G)

=>          34    156249933  diskid/DISK-5RW32T4J   GPT   (75G)
            34            6    - free -   (3.0K)
            40         1024     1  freebsd-boot   (512K)
          1064      2097152     2  freebsd-swap   (1.0G)
       2098216    154151751     3  freebsd-zfs   (74G)

[bob@pcbsd-5867] ~#
```

What this listing shows is that the third partition on **ada1** (**ada1p3**) is adequately sized to accommodate the creation of a mirror with **ada0s1a**—that is, 73G of **ada1p3** (74 GB is greater than 73 GB).

8. To create the mirror, use the `attach` subcommand of `zpool`.

```
[bob@pcbsd-5867] ~# zpool attach tank /dev/ada0s1a /dev/
ada1p3
```

Make sure to wait until resilver is done before rebooting.

If you boot from pool **tank**, you may need to update boot code on newly attached disk **/dev/ada1p3**.

Assuming you use GPT partitioning and **da0** is your new boot disk, you may use the following command:

```
gpart bootcode -b /boot/pmbr -p /boot/gptzfsboot -i 1 da0
[bob@pcbsd-5867] ~#
```

9. To check the progress of the resilvering operation, use the `zpool status` command.

```
[bob@pcbsd-5867] ~# zpool status
pool: tank
state: ONLINE
status: One or more devices is currently being resilvered.
The pool will continue to function, possibly in a degraded
state.
action: Wait for the resilver to complete.
scan: resilver in progress since Wed Oct 29 13:37:27 2014
     2.06G scanned out of 2.83G at 23.2M/s, 0h0m to go
     2.06G resilvered, 72.79% done
config:
     NAME         STATE     READ WRITE CKSUM
     tank         ONLINE      0     0     0
      mirror-0    ONLINE      0     0     0
       ada0s1a    ONLINE      0     0     0
       ada1p3     ONLINE      0     0     0(resilvering)

errors: No known data errors
```

Wait until the resilvering is complete.

EXERCISE 24.13

To test the usability of the new hard disk in the mirrored pair, shut down your machine gracefully. Then, disconnect and remove the original boot disk (in our case, **/dev/ada0**) from the machine. Finally, reboot the machine with only the new hard disk in the system. What is the status of the pool you attached the second hard disk to as a mirror after a successful reboot? After doing this exercise, you may replace the original boot disk and boot into it normally. Do the remaining step at your discretion.

10. To detach, or destroy, the two-disk mirror, use the `detach` subcommand of `zpool`.

```
[bob@pcbsd-5867] ~# zpool detach tank /dev/ada1p3
[bob@pcbsd-5867] ~#
```

Conclusion: You have created a postinstallation two-disk mirror of the system/boot disk, with another appropriately sized hard disk you have added to the system. In addition, you have used the `gpart` command to prepare that new disk for ZFS mirroring.

Along with the use of the `gpart` command to manage partitions of a new disk you have added to your system, you can also use the PC-BSD 10 Disk Manager from the Control Panel. The Disk Manager is a GUI tool that allows management of ZFS file systems, ZFS pools, and the disks on the system. It can do many of the same ZFS operations we showed as command line-based procedures in the previous example.

But to get more control over the whole range of subcommands and options of the `zpool` and `zfs` commands, and to be able to integrate that control with other UNIX commands, the command line is the more inclusive method of working with ZFS.

Example 24.5: Creating a ZFS Pool on a USB Thumb Drive for Solaris

Objectives: To use `zpool create` to create a ZFS pool on a USB thumb drive.

Introduction: The following example uses the most essential command in ZFS, the `zpool` command, with the subcommand `create` applied to a USB thumb drive on your Solaris system. The real power of ZFS as a file system/volume manager/file backup system is most obvious in multidisk computer systems, where redundancy for files and directories can be established across two or more different physical disks. In addition, this redundancy can be extended to single-disk systems with the use of the `zfs set property` command, as shown in Problem 24.9 at the end of the chapter.

Many UNIX systems with a GUI interface like Gnome or KDE (e.g., Oracle Solaris, the current Solaris family UNIX system) automatically mount the USB thumb drive on the file system, and make it available as a desktop icon. This is similar to what you would experience on non-UNIX systems.

If the only thing you want to do is use the USB thumb drive to transfer files (e.g., text files, C program source code, LibreOffice documents, and so on) to and from the computer, you do not have to use the procedures of this example to accomplish that.

If you make a mistake anywhere along the way, you can always start over by executing the cleanup step shown at the end of the example and begin again.

Prerequisites:

1. That you have a USB thumb drive that is usable on your system and has one partition and no data on it. More specifically, if the USB thumb drive is by default formatted to FAT32, then there is a high probability that it will automatically mount and show up as an icon on your Gnome desktop in Solaris. You can also type the following on the command line as superuser, to enable automounting of removable media:

   ```
   svcadm enable rmvolmgr
   ```

   If your USB thumb drive does not automount, or does not appear as an icon on the Gnome desktop in Solaris, use another USB thumb drive.
2. That there is no pool named **backup** (or any other pool) already defined on the primary partition of the USB thumb drive, and there is no pool named **test1** on your Solaris system. To find this out, type `zpool list` on the command line when the USB thumb drive has been inserted and automounts.
3. That you can determine the logical device name and the full path to it using the methods “A Quick and Easy Way to Find Out the Logical Device Names of Disks Actually Installed on Your System” from Section 24.2.1. The full path to our USB thumb drive is **/dev/rdsk/c9t0do** for Solaris. This may not be the same designation as the USB thumb drive on your system, but you can substitute your device designations for them.

Procedure: To complete the objectives of this example, do the following steps in the order presented.

1. Become the superuser.

```
bob@solaris:~$ su
Password: xxx
```

2. Put the thumb drive into a USB port on the computer, and it should automatically mount and appear as an icon on your Gnome desktop in Solaris. Determine the logical device name of the USB thumb drive, most easily done by typing the `rmformat` command on the command line, as shown in Section 24.2.1.

3. Create the pool, which we will name **backup**, on the USB thumb drive.

```
root@solaris:~# zpool create backup /dev/rdsk/c9t0d0
```

When the shell prompt reappears, the pool has been built.
Possible error messages:
   - Not designating the proper full pathname to the USB thumb drive as a vdev. To create a zpool you must designate the full path to the drive.
   - You should designate the entire disk, not just a single partition on it, as the vdev for `zpool create`.

4. List the zpools on your system.

```
root@solaris:~# zpool list
```

You should see your new zpool **backup** listed as **/backup**, along with any other pools that have been created by default. The listing for your new pool should also show the amount of disk space used on the drives, the total available disk space, and its mount point.

5. Check the listing using the `df` command.

```
root@solaris:~# df -h
```

In the listing you should see the new pool you created shown as **/backup**, right at the bottom of all the other pools created on your system by default at installation.

If you want to keep this USB thumb drive and the pool you created on it in the preceding steps so that you can do Problem 24.9 at the end of the chapter, do not continue onto the next step. If you want to reuse this USB thumb drive for another purpose, and do not want to do Problem 24.9 at the end of the chapter, proceed to the next step.

6. Now destroy the pool and exit as superuser.

```
root@solaris:~# zpool destroy backup
root@solaris:~# exit
```

Be aware that destroying the zpool will then allow you to remove the USB thumb drive from the machine. If you want to use that thumb drive for another purpose, you may have to reformat it to FAT32 and erase the data on it.

Conclusion: You have created and destroyed a zpool on a USB thumb drive. If you did not destroy it, you can now have datasets placed on it, and directories and files. All of the ZFS commands in this chapter can be used to manage zpools and datasets on this drive. You can also use this drive as a backup for user data files. See Problems 24.9 and 24.10 at the end of the chapter.

Example 24.6: Creating a ZFS Pool on a USB Thumb Drive for PC-BSD

Objectives: To create a zpool and a ZFS dataset in that pool on a thumb drive in PC-BSD.

Introduction:

The following example uses the most essential command in ZFS, the `zpool` command, with the subcommand `create` applied to a USB thumb drive on your PC-BSD system. The real power of ZFS on PC-BSD can be harnessed to common, readily available hardware vdevs that are available to place on the system by an ordinary user.

If the only thing you want to do is use a USB thumb drive to transfer files (such as text files, C program source code, LibreOffice documents, and so on) to and from the computer, <u>you do not have to use the procedures of this example to accomplish that</u>. Simply put the USB thumb drive in a USB port on the computer, and when the Mount Tray recognizes it, make the choice to mount it.

When finished transferring files, unmount it in the Mount Tray and remove it from the computer.

Many UNIX systems with a GUI interface, like Gnome or KDE, automatically mount the USB thumb drive on the file system, and make it available as a desktop icon. This is similar to what you would experience on non-UNIX systems. PC-BSD does not do this, for security and other administrative reasons. PC-BSD under KDE does make a system *tray* icon, known as the Mount Tray, available to allow you to semiautomatically mount things like USB thumb drives, and so on. It can be found in the lower-right corner of the screen, and if you right-click on it, you can make several pop-up menu choices affecting the Mount Tray.

From the KDE Kickoff Applications Launcher, you can make the **System Settings** menu choice. One of the System Settings icons is **Removable Devices**. When you click on this icon, you can turn automounting on or off, and also configure other settings for removable media. By default, automounting is off. This setting is independent of what the Mount Tray does. If you turn off automounting and quit the Mount Tray, mounting of removable media such as USB thumb drives or hard drives can be done manually.

If you make a mistake anywhere along the way, you can always start over by executing the cleanup step shown at the end of the example then begin again.

Prerequisites:

1. That you have a USB thumb drive that is usable on your system, and possibly has partitions and data on it. More specifically, if the USB thumb drive is by default formatted to FAT32, then there is a high probability that it will be recognized by the PC-BSD Mount Tray.

   If your USB thumb drive is not recognized by the Mount Tray, use another USB thumb drive.
2. That you can determine the logical device name and the full path to it using the methods "A Quick and Easy Way to Find Out the Logical Device Names of Disks Actually Installed on Your System" from Section 24.2.1. The full path to our USB thumb drive is **/dev/da0** for PC-BD. This may not be the same designation as the USB thumb drive on your system, but you can substitute your device designations for them.
3. That you have done Example 23.5 in Chapter 23, Section 23.4.3.

Procedure: All commands are shown in **bold** type and are typed at the command line.

1. Put the thumb drive into the USB port on the computer. If it is FAT32 formatted, it will be recognized by the Mount Tray. Do not mount it with the Mount Tray facility!

2. Open a console window and type the following in it to become the superuser:

```
[bob@pcbsd-4382] ~% su
Password: xxx
root@pcbsd-4382:/usr/home/bob #
```

3. Launch the Disk Manager with a GUI. You will use the Disk Manager to easily delete partitions from the thumb drive, and destroy its partition table. So make sure there is no data you want to keep on the thumb drive.

```
root@pcbsd-4382:/usr/home/bob # pc-zmanager
Locale: "en"
Disk Manager launches.
```

4. In the **Disks** tab, right-click on any of the partitions that exist on **da0** (the thumb drive itself should appear as **da0**, and any partitions should appear as **da0p1**, **da0p2**, etc.), and make the choice **Destroy This Slice**. Do this for all partitions. If none exist, go to the next step.

5. In the **Disks** tab, right-click on the thumb drive (**da0**) and make the choice **Delete Partition Table**.

6. In the **Disks** tab, right-click on the thumb drive (**da0**) and make the choice **Create GPT partition table**. Now the thumb drive should be listed as "Available."

7. Kill the Disk Manager by clicking on the kill window button in the upper-right corner of the Disk Manager GUI window.

8. You will now use the `gpart` command to create a primary partition on the thumb drive. The file system created on this partition will be a FreeBSD ZFS file system. It will be added as **da0p1**.

```
root@pcbsd-4382:/usr/home/bob # gpart add -t freebsd-zfs
                                -l zfs1 /dev/da0
da0p1 added
```

9. You can now create a zpool named **test3** on this new primary partition on the thumb drive.

```
root@pcbsd-4382:/usr/home/bob # zpool create test3/dev/
da0p1
```

10. You can now create a file system named **newfilesystem** on the zpool **test3**.

```
root@pcbsd-4382:/usr/home/bob # zfs create test3/
newfilesystem
```

11. Obtain a listing of the datasets on your computer. Several datasets will be listed, including your home directory. Notice in the listing that the mount point of each dataset is given in the last column. This is the path that you designate to the datasets' directory. If you put files in that directory, they are part of that file system.

```
root@pcbsd-4382:/usr/home/bob # zfs list
NAME                 USED  AVAIL REFER MOUNTPOINT
output truncated...
test3                81.5K 7.02G 19K   /test3
test3/newfilesystem 19K   7.02G 19K   /test3/newfilesystem
```

To put files in the ZFS dataset **newfilesystem**, put them in the directory **/test3/newfilesystem**.

12. If you want to continue using this thumb drive as an additional drive on your system, stop here. Be aware that if you remove the thumb drive without unmounting it, unexpected results will occur. If you want to use the thumb drive for other purposes, continue.

    The following three steps allow you to undo everything you have done in this example (except, of course, the deletion of any data that was on the thumb drive before you began this example!).

13. To begin, first destroy the file system.

```
root@pcbsd-4382:/usr/home/bob # zfs destroy test3/
newfilesystem
```

14. Then destroy the zpool.

```
root@pcbsd-4382:/usr/home/bob # zpool destroy test3
```

15. Launch the Disk Manager as you did in step 3, and in the **Disks** tab, destroy the slice **da0p1** by right-clicking and making the choice **Destroy This Slice**. Check in the ZFS **Pools** tab that there is no longer any record of **test3**.

16. Pull the thumb drive out of the USB port.

Conclusion: You have created and destroyed a zpool and a ZFS dataset on a USB thumb drive that you attached to your PC-BSD machine. If you did not destroy the zpool and dataset on it, you can now create additional datasets on it and place files in those datasets. All of the ZFS commands in this chapter can be used to manage zpools and datasets on this drive. You can also use this drive as a backup for user data files.

24.3 ZFS COMMANDS AND OPERATIONS

The following section is an abbreviated encyclopedia, or reference manual, that illustrates many uses of the two important ZFS commands, `zfs` and `zpool`. It shows the kinds of operations you can perform with those two commands and with their options and subcommands. In order to get a complete listing, with examples, of the commands, subcommands, and options, consult the man pages for `zfs` or `zpool` on your system! The Command Appendix on the CRC website contains entries for `zfs` and `zpool`. Those entries show more detail about options and subcommands.

We first present a summary of the command categories and basic definitions for `zpool` and `zfs`. We then show several examples of `zpool` and `zfs` command, subcommand, option, and command argument usage.

This section assumes that you have done at least one or more of the previous examples in Section 24.2 to get a feel for what ZFS can do. The command syntax shown in this section is valid for both PC-BSD and Solaris, and any differences between implementations on those two systems will be duly noted by appending the applicable system name in parentheses to the command in question.

All sample code you type on the command line is shown in **`bold`** text and is always followed by pressing `<Enter>` on the keyboard. Comments specific to a command, operation, or term usually appear <u>after</u> the item of interest.

24.3.1 Command Categories and Basic Definitions

1. Directories and Files

   Where error messages appear: **/var/adm/messages** (Solaris), **console**

2. ZFS States

   DEGRADED One or more top-level devices is in the degraded state because they have become offline. Sufficient replicas exist to keep functioning.

   FAULTED One or more top-level devices is in the faulted state because they have become offline. Insufficient replicas exist to keep functioning.

   OFFLINE The device was explicitly taken offline by the `zpool offline` command.

ONLINE The device is online and functioning.
REMOVED The device was physically removed while the system was running.
UNAVAIL The device could not be opened.

3. Scrubbing and Resilvering
   Scrubbing: Examines all data to discover hardware faults or disk failures. Only one scrub may be running at one time, and you can manually scrub.
   Resilvering: The same concept as rebuilding or resyncing data on to new disks into an array. The smart thing resilvering does is it does not rebuild the whole disk, only the data that is required (the data blocks not the free blocks), thus reducing the time to *resync* a disk. Resilvering is automatic when you replace disks and so on. If a scrub is already running, it is suspended until the resilvering has finished, then the scrubbing will continue.

4. ZFS Devices and Device Terminology
   Disk: A physical disk drive.
   File: The absolute path of preallocated files/images.
   Mirror: Standard RAID1 mirror.
   Raidz1/2/3: Nonstandard distributed parity-based software RAID levels. Basically, if a power failure occurs in the middle of a write then you have the data plus the parity, or you don't. Also, ZFS supports *self-healing*, which means that if it cannot read a bad block it will reconstruct it using the parity, and repair or indicate that this block should not be used.
      Raidz1: 3, 5, 9 disks
      Raidz2: 4, 6, 8, 10, 18 disks
      Raidz3: 5, 7, 11, 19 disks
   The more parity bits, the longer it takes to resilver an array. Standard mirroring does not have the problem of creating the parity, so it is quicker in resilvering. Raidz is more like RAID3 than RAID5, but does use parity to protect from disk failures.
      Raidz/Raidz1: A minimum of three devices (one parity disk); you can suffer a one-disk loss.
      Raidz2: A minimum of four devices (two parity disks); you can suffer a two-disk loss.
      Raidz3: A minimum of five devices (three parity disks); you can suffer a three-disk loss.
   Spare: hard drives marked as *hot spare* for ZFS RAID. By default, hot spares are not used in a disk failure; you must turn on the *autoreplace* feature.

5. Cache
   A zfs cache caches both the least recently used (LRU) and least frequently used (LFU) block requests; the cache device uses level-2 adaptive read cache (L2ARC).

6. Log
   There are two log types used:

ZFS intent log (ZIL): A logging mechanism where all the data to be written is stored, then later flushed, as a transactional write; this is similar to a journal file system (**ext3** or **ext4**).

Separate intent log (SLOG): A separate logging device that caches the synchronous parts of the ZIL before flushing them to the slower disk; it does not cache asynchronous data (asynchronous data is flushed directly to the disk). If the SLOG exists, the ZIL will be moved to it rather than residing on the platter disk; everything in the SLOG will always be in the system memory. Basically, the SLOG is the device and the ZIL is data on the device.

24.3.2 ZFS Storage Pools and the `zpool` command

The subcommands and options shown in this section are presented in this general way, with Solaris syntax shown by default. PC-BSD syntax is enclosed in `( )`.

x. What the command, subcommand, and options accomplish.

```
The command, subcommand, options, and command arguments
```

Commentary or explanation.
Further examples:

```
More variations of the command, subcommand, options and command
arguments
```

Commentary or explanation.

1. How to display zpools:

   ```
   zpool list
   ```

   Further examples:

   ```
   zpool list -o poolname, size, altroot
   ```

   There are a number of properties that you can select, the default is: `name`, `size`, `used`, `available`, `capacity`, `health`, `altroot`.

2. How to display zpool status:

   ```
   zpool status
   ```

   Further examples:

   ```
   zpool status -xv
   ```

   Shows only errored pools with more verbosity.

3. How to show zpool statistics:

   ```
   zpool iostat -v 5 5
   ```

   Use this command like you would `iostat`

4. How to show zpool history:

```
zpool history -il
```

Once a pool has been removed, the history is gone!

5. How to create a zpool:

```
zpool create -n data2 /dev/dsk/c1t0d0s0
```

You cannot shrink a pool, only grow it!
The `-n` option performs a dry run but doesn't actually perform the creation.
Further examples:

```
zpool create data2 /dev/ada1p1 /dev/ada2p1
```

Assumes there are two disks called **/dev/ada1p1** and **/dev/ada2p1** (PC-BSD).

```
zpool create data2a /dev/dsk/c1t0d0s0
```

Using a standard disk slice on **c1t0d0**, slice **s0**.

```
zpool create -m /zfspool data2a /dev/dsk/c1t0d0s0
```

Using a different mount point than the default **/<pool name>**.

```
zpool create data3 mirror c1t0d0 c2t0d0 mirror c1t0d1 c2t0d1
zpool create data4 mirror c1t0d0 c2t0d0 spare c3t0d0
```

Mirror and hot spare disk examples; hot spares are not used by default, you need to turn on the autoreplace feature with `zpool set autoreplace=on` for each pool.

```
zpool create data5 mirror c1t0d0 c2t0d0 log mirror c3t0d0
    c4t0d0
```

Setting up a log device and mirroring it.

```
zpool create data6 mirror c1t0d0 c2t0d0 cache c3t0d0 c3t1d0
```

Setting up a cache device.

```
zpool create data7 raidz2 c1t0d0 c1t1d0 c1t2d0 c1t3d0 c1t4d0
```

You can also create RAID pools (RAIDZ/RAIDZ1: mirror; RAIDZ2: single parity; RAIDZ3: double parity).

6. How to destroy a zpool:

```
zpool destroy data2
```

Further examples:

```
zpool import -f -D -d /tank/data2
```

You can reimport a destroyed pool.

7. How to add a device to a zpool:

```
zpool add data01 c2t0d0
```

zpool only supports the removal of hot spares and cache disks! Therefore, be sure you want to add the device to the pool, because you cannot ordinarily remove it with the `zpool remove` command. For adding to mirrors, see the `attach` and `detach` subcommands that follow.

8. How to resize a zpool:
   When replacing a disk with a larger one you must enable the *autoexpand* feature to allow you to use the extended space. You must do this before replacing the first disk.

9. How to remove a zpool:

   ```
   zpool remove data01 c2t0d0
   ```

   `zpool` only supports the removal of hot spares and cache disks! Therefore, be sure you want to add the device to the pool, because you cannot ordinarily remove it with the `zpool remove` command. For adding to mirrors, see the `attach` and `detach` subcommands that follow.

10. How to clear faults:

    ```
    zpool clear data01
    ```

    Further examples:

    ```
    zpool clear data01 c2t0d0
    ```

    Clears a specific disk fault.

11. Attaching additional drives as a mirror:

    ```
    zpool attach data01 c2t0d0 c3t0d0
    ```

    c2t0d0 is an existing disk that is not mirrored, so by attaching **c3t0d0** to the pool **data01**, both disks will become a mirrored pair.

12. How to detach a mirror disk:

    ```
    zpool detach data01 c2t0d0
    ```

    See the previous note on attaching additional drives as a mirror.

13. How to *online* a zpool (put the pool online):

    ```
    zpool online data01 c2t0d0
    ```

14. How to *offline* a zpool (take the pool offline):

    ```
    zpool offline data01 c2t0d0
    ```

    Further examples:

    ```
    zpool offline data01 -t c2t0d0
    ```

    This achieves temporary offlining using `-t` (will revert back to online after a reboot).

15. How to replace pools:

    ```
    zpool replace data03 c2t0d0
    ```

    Replaces one disk that uses the same designation in **/dev** as another disk.
    Further examples:

    ```
    zpool replace data03 c2t0d0 c3t0d0
    ```

    Replaces one disk with another disk in **/dev** that has a different designation.

16. How to do scrubbing:

    ```
    zpool scrub data01
    ```

    Further examples:

    ```
    zpool scrub -s data01
    ```

    Stop a scrubbing in progress; check the scrub line using `zpool status data01` to see any errors.

17. How to do exporting:

    ```
    zpool export data01
    ```

    You can list exported pools using the import command `zpool import` to find what the names of exported zpools are, if any.

18. How to do importing:

    ```
    zpool import data01
    ```

    When using standard disk devices—that is, **c2t0d0**.
    Further examples:

    ```
    zpool import -d /zfs
    ```

    If using files in the **/zfs** file system

    ```
    zpool import -f -D -d /zfs1 data2
    ```

    Imports a destroyed pool.

19. Getting zpool parameters:

    ```
    zpool get all data01
    ```

    The source column denotes if the value has been changed from its default value; a dash in this column means it is a read-only value.

20. Setting zpool parameters:

    ```
    zpool set autoreplace=on data01
    ```

    Use the command `zpool get all <pool>` to obtain a list of current settings.

21. How to upgrade pools:

    ```
    zpool upgrade -v
    ```

Lists upgrade paths.
Further examples:

```
zpool upgrade -a
```

Upgrades all pools.

```
zpool upgrade data01
```

Upgrades a specific pool; use `zpool get all poolname` to obtain the version number of a pool.

```
zpool upgrade -V 10 data01
```

Upgrades to a specific version.

22. Replace a failed disk:

    ```
    zpool list
    ```

    Lists the zpools and identifies the failed disk.
    Further examples:

    ```
    zpool replace data01 c1t0d0
    zpool replace data01 c1t0d0 c1t1d0
    ```

    Replaces the disk (you can use same disk or a new disk of equal or larger capacity).

    ```
    zpool clear data01
    ```

    Clears any existing errors.

    ```
    zpool scrub data01
    ```

    Scrub the pool to check for any more errors (this depends on the size of the zpool, as it can take a long time to complete). You can now remove the failed disk in the normal way, depending on your hardware.

23. How to expand a pool's capacity:

    ```
    zpool replace data01 c1t0d0 c2t0d0
    zpool set autoexpand=on data01
    ```

    You cannot remove a disk from a pool and you cannot shrink the pool, but you can enlarge it by replacing existing disks with larger disks!

24. How to use the **`monitor`** subcommand (as of Solaris 11.3):

SYNTAX

```
zpool monitor -t provider [-T d|u] [[-p] -o field[,…]] [pool] …
[interval [count]]
```

`monitor` displays status or progress information for the given pools. If no pool is entered, information for all pools is displayed. When given an **`interval`**, the information is printed every **`interval`** seconds until `<Ctrl+C>` is pressed. If **`count`** is specified, the command exits after **`count`** reports are printed.

Options:

`-o field[,...]`	Display only selected field(s).
`-t provider`	Display data from the listed providers. Current providers are `send`, `receive` (or `recv`), `destroy`, `scrub`, and `resilver`. An up-to-date list of providers is available if you give the command `zpool help monitor`.
`-T d|u`	Display a time stamp. Specify **`d`** for standard date format. See **`date`**. Specify **`u`** for a printed representation of the internal representation of time. See **`time`**.

Example: Obtaining parsable output
The following command is used to obtain parsable output and will provide one interval.
# **zpool monitor -p -o pool, pctdone, other -t send poolA poolC**

```
poolA:20.4:poolA/fs2/team2@fs2 _ all
poolA:0.0:poolA/fs2/team2@all
poolA:28.6:poolA/fs1/team3@fs1 _ all
poolC:33.3:poolC/fs1/team2@fs1 _ all
poolC:50.0:poolC/fs2/team1@fs2 _ all
```

24.3.3 ZFS File System Commands and the `zfs` Command

The subcommands and options shown in this section are presented in the following general way, with Solaris syntax shown by default.

x. What the command, subcommand, and options accomplish.

```
The command, subcommand, options, and command arguments
```

Commentary or explanation.
Further examples:

```
More variations of the command, subcommand, options and
command arguments
```

Commentary or explanation.

1. Displaying ZFS file systems:

```
zfs list
```

Lists all ZFS file systems
Further examples:

```
zfs list -t filesystem
zfs list -t snapshot
zfs list -t volume
zfs list -t all -r poolname
```

Lists different types (`file system`, `snapshot`, `volume`) by **poolname**.

```
zfs list -r data01/sarwar
```

Recursive display.

```
zfs list -o poolname,mounted,sharenfs,mountpoint
```

Complex listing: there are a number of attributes that you can use in a complex listing; see the man page for `zfs`.

2. How to create a file system:

   ```
   zfs create data01/sarwar
   ```

   Presumes a pool called **data01**, creates a **/data01/sarwar** ZFS file system.
   Further examples:

   ```
   zfs create -o mountpoint=/users/data01/users
   ```

   Creates at a different mount point.

3. How to destroy a file system:

   ```
   zfs destroy data01/sarwar
   ```

   Further examples:

   ```
   zfs destroy -r data01/sarwar
   zfs destroy -R data01/sarwar
   ```

   Uses the recursive options `-r` (all children), `-R` (all dependents).

4. How to mount a file system:

   ```
   zfs mount data01
   ```

   Further examples:

   ```
   zfs mount -o mountpoint=/tmpmnt data01/sarwar
   ```

   You can create temporary mount that expires after unmounting. You can apply all the normal mount options—that is, `ro`/`rw`, `setuid`, and so on.

5. How to unmount a file system:

   ```
   zfs umount data01
   ```

6. How to share a file system:

   ```
   zfs share data01
   ```

   Further examples:

   ```
   zfs set sharenfs=on data01
   ```

   This file system persists after reboots!

   ```
   zfs set sharenfs="rw=@192.168.0.13/24" data01/sarwar
   ```

   Shares with specific hosts.

7. How to unshare a file system:

   ```
   zfs unshare data01
   ```

   Further examples:

```
zfs set sharenfs=off data01
```

This file system persists after reboot!

8. How to take snapshots of file systems:
   Taking a “snapshot” of a file system is like taking a picture: changes are recorded to the snapshot when the original file system changes; to remove a dataset all previous snapshots have to be removed. You can also rename snapshots. You cannot destroy a snapshot if it has a clone.

   ```
   zfs snapshot data01@10022010
   ```

   Creates a snapshot.
   Further examples:

   ```
   zfs snapshot data01@10022010 data01@mybackup
   ```

   Renames a snapshot.

   ```
   zfs destroy data01@10022010
   ```

   Destroys a snapshot.

9. How to roll back a file system:
   By default, you can only roll back to the latest snapshot. To roll back to older ones, you must delete all newer snapshots.

   ```
   zfs rollback data01@10022010
   ```

10. Cloning/promoting file systems:
    Clones are writable file systems that have been upgraded from a snapshot. A dependency will remain on the snapshot as long as the clone exists. A clone uses the data from the snapshot to exist. As you use the clone, it uses space separate from the snapshot. Clones cannot be created across zpools, you need to use the `zfs send/receive` commands to do this, as shown in Example 24.4.

    ```
    zfs clone data7@10022010 data8/clone
    zfs clone -o mountpoint=/clone data7@10022010 data8/clone
    ```

    Clones, changes the mount point of the clone.
    Further examples:

    ```
    zfs promote data8/clone
    ```

    Promotes a clone. This allows you to destroy the original file system that the clone is attached to. The clone must reside in the same pool!

11. Renaming a file system:

    ```
    zfs rename data03/koretsky_disk01 data03/koretsky_d01
    ```

    The dataset must be kept within the same pool. There are two options on this command: `-p` creates all the nonexistent parent datasets; `-r` recursively rename the snapshots of all descendant datasets (used with snapshots only).

12. Compression of file systems:

```
zfs set compression=lzjb data03/sarwar
```

You enable compression by setting a feature. Compressions are `on`, `off`, `lzjb`, `gzip`, `gzip[1–9]` and `zle`. Compression only starts when you turn it on; other existing data will not be compressed.
Further examples:

```
zfs get compressratio data03/sarwar
```

You can get the compression ratio.

13. Deduplication:
You can save disk space using *deduplication*, which can be on the level of file, block, or byte. For example, at the file level, each file is hashed with a cryptographic hashing algorithm such as `SHA-256`; if two files match, then just point to the existing file rather than storing a whole new file. This is ideal for small files, but for large files, a single character change would mean that all the data has to be copied over again! Block deduplication allows you to share all the same blocks in a file minus the blocks that are different; this allows the sharing of unique blocks on disk and the reference-shared blocks in RAM. However, a lot of RAM is necessary to keep track of which blocks are shared and which are not. This is the preferred option rather than file or byte deduplication. Shared blocks are stored in what is called a *deduplication table*; the more deduplicated blocks there are, the larger the table. The table is read every time to make a block change, thus the table should be held in fast RAM. If you run out of RAM, then the table will be saved onto disk. So how much RAM do you need? You can use the `zdb` command to check and take the *bp count*. A good rule of thumb is to allow 5 GB of RAM for every 1 TB of disk.

```
zdb -b data01
```

Use this command to see the block the dataset consumes.
Further examples:

```
zfs set dedup=on data01/myfiles
```

To turn on deduplicate.

```
zfs get dedupratio data01/myfiles
```

To see the deduplication ratio.

```
zdb -DD poolname
```

To see a histogram of how many blocks are referenced how many times.

14. Getting file system parameters:

```
zfs get all data03/sarwar
```

Lists all the properties.
Further examples:

```
zfs get setuid data03/sarwar
```

Gets a specific property.

```
zfs get compression
```

Gets a list of a specific properties for all datasets. The source column denotes if the value has been changed from its default value; a dash in this column means it is a read-only value.

15. Setting file system parameters:

```
zfs set copies=2 data03/sarwar
```

Sets the number of copies of dataset **sarwar** in the pool **data03** to 2; the default number of copies is 1. This is probably the most useful and important way to ensure redundancy on a nonredundant vdev, such as a single hard disk in a laptop computer. Although it doubles the storage space required to contain the dataset, error correction with `zpool scrub` can be achieved on the nonredundant vdev that contains the pool and its datasets that have copies set to 2.
Further examples:

```
zfs set quota=50M data03/sarwar
zfs set quota=none data03/sarwar
```

Sets and unsets the disk usage quota. Use the command `zfs get all <dataset>` to obtain a list of current settings.

16. How to have a file system inherit attributes:

```
zfs inherit compression data03/sarwar
Sets back to the default value.
```

17. How to upgrade the ZFS version:

```
zfs upgrade –v
```

Lists the upgrade paths.
Further examples:

```
zfs upgrade
```

Lists all the datasets that are not at the current level.

```
upgrade -V <version> data03/oracle
```

Upgrades a specific dataset.

18. How to use allow/unallow:

```
zfs allow master
```

Displays the permissions set and any user permissions.
Further examples:

```
zfs allow -s @permset1 create,mount,snapshot,clone,promote
   master
```

Creates a permission set.

```
zfs unallow -s @permset1 master
```

Deletes a permission set.

```
zfs allow vallep @permset1 master
```

Grants a user permissions.

```
zfs unallow vallep @permset1 master
```

Revokes a user's permissions. There are many permissions that you can set. Refer to the `zfs` man page, or just use the `zfs allow` command, to get help.

24.4 FILE SYSTEM BACKUPS USING `ZFS SNAPSHOT`

Snapshots are the ZFS way of creating archives and backups automatically or with very simple operations and commands. As stated previously, taking a snapshot of a file system is like taking a picture; changes are recorded to the snapshot when the original file system changes.

Here are some important things to remember about snapshots:

- To remove a dataset, all previous snapshots have to be removed.
- You can rename snapshots.
- You cannot destroy a snapshot if it has a clone.

24.4.1 Examples of snapshot

An example of creating a snapshot is:

```
zfs snapshot data01@10022010
```

An example of renaming a snapshot is:

```
zfs snapshot data01@10022010 data01@mybackup
```

An example of destroying a snapshot is:

```
zfs destroy data01@10022010
```

24.4.2 `zfs rollback`

It is possible to roll back a file system, or return it to a previous state. You must use the `zfs rollback command`. By default you can only roll back to the latest snapshot, to roll back to an older one you must delete all newer snapshots!

An example of rolling back to a snapshot is:

```
zfs rollback data01@10022010
```

24.4.3 Cloning/Promoting

As stated previously, clones are writable file systems that have been upgraded from a snapshot, and a dependency will remain on the snapshot as long as the clone exists. A clone uses the data from the snapshot to exist. As you use the clone it uses space separate from the snapshot. Clones cannot be created across zpools, you need to use the `zfs send/receive` commands to do this.

Two examples of cloning are:

```
zfs clone data7@10022010 data8/clone
zfs clone -o mountpoint=/clone data7@10022010 data8/clone
```

Promoting a clone allows you to destroy the original file system that the clone is attached to. An example of this is:

```
zfs promote data8/clone
```

The clone must reside in the same pool.

24.4.4 Renaming a Filesystem

The dataset must be kept within the same zpool! An example of this is:

```
zfs rename data03/koretsky_disk01 data03/koretsky_d01
```

There are two options on this command: `-p` creates all the nonexistent parent datasets; `-r` recursively renames the snapshots of all descendant datasets (used with snapshots only).

24.4.5 Compression of Filesystems

You enable compression by setting a feature. Compressions are `on`, `off`, `lzjb`, `gzip`, `gzip[1–9]` and `zle`. Compression starts when you turn it on; other existing data will not be compressed. An example of this is:

```
zfs set compression=lzjb data03/sarwar
```

You can get the compression ratio by using the following example:

```
zfs get compressratio data03/sarwar
```

24.4.6 Bourne Shell Script Example for Incremental ZFS Backups

Example 24.7: `zfs snapshot` Command Automation in a Bourne Shell Script

The following Bourne shell script achieves the incremental backing up of a file system on one computer to a remote host system, using `zfs snapshot send/receive`. It is very similar to, and a further extension of, the `zfs send/receive` example code shown previously.

The prerequisites for running this script are:

1. The host receiving the snapshot must be running the same or a higher version of ZFS than the sender.
2. You must have root access on the host receiving the backup.
3. You must be sending to an account that has ZFS create/receive properties.
4. You have previously created a snapshot of the source.
5. You have previously created a Z file system on the destination named **/tank/test**.

```
#!/bin/sh
# This assigns a local file system as the source to be
# transmitted
pool="/usr/src"
# This assigns a remote destination
# destination="tank/test"
# This names the IP address of the remote target host
# host="192.168.0.7"
# This sets the date format for today in PC-BSD
# For Solaris the date format should be today = 'date +
"%Y-%m-%d"'
# today='date +"$type-%Y-%m-%d"'
# This sets the date format for yesterday in PC-BSD
# For Solaris the date format should be yesterday = 'date
# -d"-1 day" + "%Y-%m-%d"'
yesterday='date -v -1d +"$type-%Y-%m-%d"'
# Create today's snapshot
snapshot_today="$pool@$today"
# Look for a snapshot with this name, and if none exists,
# take the snapshot
if zfs list -H -o name -t snapshot | sort | grep "$snapshot_
today$" > /dev/null
   then
       echo " snapshot, $snapshot_today, already exists"
       exit 1
   else
       echo " taking todays snapshot, $snapshot_today"
       zfs snapshot -r $snapshot_today
fi
# Look for yesterday's snapshot
snapshot_yesterday="$pool@$yesterday"
# If it exists, ZFS sends today's snapshot
if zfs list -H -o name -t snapshot | sort | grep "$snapshot_
yesterday$" > /dev/null
   then
       echo " yesterday snapshot, $snapshot_yesterday,
exists, send todays backup"
       zfs send -R -i $snapshot_yesterday $snapshot_today |
ssh root@$host zfs receive -Fduv\          $destination
      echo " backup complete destroying yesterdays snapshot"
      zfs destroy -r $snapshot_yesterday
      exit 0
   else
     echo " missing yesterday snapshot aborting,
$snapshot_yesterday"
      exit 1
fi
```

24.5 USING ACCESS CONTROL LISTS (ACLS) AND ATTRIBUTES FOR SECURING SOLARIS ZFS FILES

In contrast to the traditional UNIX permissions model, which defines secure access to an object like a file or directory via permissions like read, write, and execute, the *access control list* (ACL) model gives the user finer-grained base control over object security.

In the permissions model, group permissions are the only way by which a file owner can relegate access to different constituencies of the user base. That is because any file can only belong to one group. Therefore, to serve different constituencies, many different groups have to exist. Only administrators can create and assign group membership. And sharing of files between collaborative working project teams becomes untenable using the permissions model.

ACLs provide greater discretionary power, but at the cost of more complexity, larger storage requirements, and slower performance of the underlying file system, whether that be VFS or ZFS. Our two UNIX base systems, PC-BSD and Solaris, with ZFS, support the permissions model and the ACL model (referred to here as NFSv4). Since there are several ACL models, it is worth noting here that ZFS only supports NFSv4 ACLs.

We showed the permissions model in Chapter 5, Sections 5.4 and 5.5. The methods shown in this section apply to Solaris ZFS files. In Chapter 23, Section 9.3, we detailed a similar method of managing ACLs for ZFS files and directories on a PC-BSD system.

We cover the following topics in the subsections indicated:

24.5.1 Solaris ACL Model
24.5.2 Setting ACLs on ZFS Files and Command Syntax for Setting ACLs
24.5.3 Setting ACL Inheritance on ZFS Files

24.5.1 Solaris ACL Model

The Solaris ACL model fully supports the interoperability that NFSv4 offers between UNIX and non-UNIX clients. It is similar to Windows NT–style ACLs, and provides more detailed access control than is available with standard file permissions. These ACLs are set and displayed with the `chmod A` and `ls -(v, V, dv, dV)` commands.

The ACL model has two types of *access control entries* (ACEs) that affect access checking: `ALLOW` and `DENY`. Therefore, you cannot infer from any single ACE that defines a set of permissions whether the permissions that are not defined in that ACE are allowed or denied.

24.5.1.1 ACL Formats

ACLs have two basic formats:

1. Trivial ACL: Contains only entries for traditional UNIX user categories that are represented as `owner@`, `group@`, and `everyone@`.

   For a newly created file, the default ACL has the following entries:

   ```
   0:owner@:read_data/write_data/append_data/read_xattr/
   write_xattr
   ```

```
/read_attributes/write_attributes/read_acl/write_acl/
write_owner
/synchronize:allow
1:group@:read_data/read_xattr/read_attributes/read_acl/
synchronize:allow
2:everyone@:read_data/read_xattr/read_attributes/read_acl/
synchronize
:allow
```

For a newly created directory, the default ACL has the following entries:

```
0:owner@:list_directory/read_data/add_file/write_data/
add_subdirectory
/append_data/read_xattr/write_xattr/execute/delete_child
/read_attributes/write_attributes/read_acl/write_acl/
write_owner
/synchronize:allow
1:group@:list_directory/read_data/read_xattr/execute/
read_attributes
/read_acl/synchronize:allow
2:everyone@:list_directory/read_data/read_xattr/execute/
read_attributes
/read_acl/synchronize:allow
```

2. Nontrivial ACL: Contains entries for added user categories. The entries might also include inheritance flags or are ordered in a nontraditional way. A nontrivial entry might look like the following example, where permissions are specifically granted to user **mansoor**.

```
0:user:mansoor:read_data/write_data:file_inherit:allow
```

24.5.1.2 ACL Entry Descriptions

Components of NFSv4 ACL command specification for Solaris:

```
Files       chmod A0+user:bob:rwx-----------:-------:allow filename
              a   bcd  e   f         g          h     i       j
Directories chmod A0+user:bob:r-------------:fd-----:allow dirname
              a   bcd  e   f         g          h     i       j
```

Key:

a	chmod command that sets ACL entries
b	Option A
c	Position option argument starting at 0; also serves as the index of the entry
d	Operator option argument, + to add, = to replace
e	ACL tag, in these cases **user**, the first part of the ACL entry type
f	ACL qualifier, in these cases **bob**, the second part of the ACL entry type
g	14 permissions, shown in short form, the rest optional
h	7 inheritance flags, for directories only, the rest optional

i ACL type, `allow` or `deny`
j Command argument, a filename or directory name specification

Format of `ls -V` output for the preceding files command

```
%ls -V filename
traditional permissions header
            user:bob:rwx-----------:-------:allow filename
              owner@ -------------default------------------
              group@ --------------default-----------------
            everyone@ -----------default---------------------
```

Use the following sample entry as a reference to understand the elements that comprise an ACL entry. These elements apply to both trivial and nontrivial ACLs.

```
0:user:mansoor:read_data/write_data:file_inherit:allow
```

Index: A number at the beginning of the entry, such as the number zero (0) in the example. The index identifies a specific entry and distinguishes the entry from others in the ACL. The index appears in the long format output of ACL entries given by the `ls –V` command.

ACL entry type: The user category. In trivial ACLs, only entries for `owner@`, `group@`, and `everyone@` are set. In nontrivial ACLs, `user:username` and `group:groupname` are added. In the example, the entry type is `user:mansoor`.

Access privileges: Permissions that are granted or denied to the entry type. In the example, user **mansoor**'s permissions are `read _ data` and `write _ data`.

Inheritance flags: An optional list of ACL flags that control how permissions are propagated downward in a directory structure. In the sample entry, `file _ inherit` is also granted to user **mansoor**.

Permission control type: Determines whether the permissions in an entry are allowed or denied. In the example, the permissions for **mansoor** are allowed.

Table 24.1 describes each ACL entry type.
Table 24.2 describes ACL access privileges.

TABLE 24.1 ACL Entry Types

ACL Entry Type	Description
`owner@`	Specifies the access granted to the owner of the object
`group@`	Specifies the access granted to the owning group of the object
`everyone@`	Specifies the access granted to any user or group that does not match any other ACL entry
`user`	With a user name, specifies the access granted to an additional user of the object
`group`	With a group name, specifies the access granted to an additional group of the object

TABLE 24.2 ACL Access Privileges

Access Privilege	Compact Access Privilege	Description
`add_file`	`w`	Permission to add a new file to a directory
`add_subdirectory`	`p`	On a directory, permission to create a subdirectory
`append_data`	`p`	Permission to modify a file but only beginning from the EOF
`delete`	`d`	Permission to delete a file
`delete_child`	`D`	Permission to delete a file or directory within a directory
`execute`	`x`	Permission to execute a file or search the contents of a directory
`list_directory`	`r`	Permission to list the contents of a directory
`read_acl`	`c`	Permission to read the ACL (`ls`)
`read_attributes`	`a`	Permission to read basic attributes (non-ACLs) of a file
`read_data`	`r`	Permission to read the contents of the file
`read_xattr`	`R`	Permission to read the extended attributes of a file
`synchronize`	`s`	Permission to access a file locally at the server with synchronized read and write operations
`write_xattr`	`W`	Permission to create extended attributes or write to the extended attributes directory
`write_data`	`w`	Permission to modify or replace the contents of a file
`write_attributes`	`A`	Permission to change the times associated with a file or directory to an arbitrary value
`write_acl`	`C`	Permission to write the ACL or the ability to modify the ACL by using the `chmod` command
`write_owner`	`o`	Permission to change the file's owner or group; or the ability to execute the `chown` or `chgrp` commands on the file

TABLE 24.3 ACL `delete` and `delete_child` Permission Behavior

Parent Directory Permissions	Target Object Permissions		
" " (empty)	ACL Allows Delete	ACL Denies Delete	Delete Permission Unspecified
ACL allows `delete_child`	Permit	Permit	Permit
ACL denies `delete_child`	Permit	Deny	Deny
ACL allows only write and execute	Permit	Permit	Permit
ACL denies write and execute	Permit	Deny	Deny

Table 24.3 provides additional details about ACL `delete` and `delete _ child` behavior.

24.5.1.2.1 ZFS ACL Sets

An ACL set consists of a combination of ACL permissions. These ACL sets are predefined and cannot be modified.

ACL Set Name	Included ACL Permissions
`full _ set`	All permissions

`modify_set`	All permissions except `write_acl` and `write_owner`
`read_set`	`read_data`, `read_attributes`, `read_xattr`, `read_acl`
`write_set`	`write_data`, `append_data`, `write_attributes`, `write_xattr`

You can apply an ACL set rather than having to set individual permissions separately. For example, granting **mansoor** the `read_set` ACL set gives him permission to read ACLs as well as file contents and their basic and extended attributes.

```
# chmod A+user:mansoor:read_set:allow file.0
# ls -V file.0
-r--r--r--+ 1 root root 206695 Jul 20 13:43 file.0
0:user:mansoor:read_data/read_xattr/read_attributes/read_acl:allow
...
```

24.5.1.3 ACL Inheritance

ACL inheritance means that a newly created file or directory can inherit the ACLs they are intended to inherit without disregarding the existing permission bits on the parent directory. By default, ACLs are not propagated. If you set a nontrivial ACL on a directory, it is not inherited to any subsequent directory. You must specify the inheritance of an ACL on a file or directory.

Table 24.4 describes the optional inheritance flags.

In addition, you can set a default ACL inheritance policy on the file system that is more strict or less strict by using the ZFS `aclinherit` file system property. For more information about this property, see Section 24.5.1.4, "ACL Properties."

For more information about setting ACL inheritance on ZFS files, see Section 24.5.3, "Setting ACL Inheritance on ZFS Files."

TABLE 24.4 ACL Inheritance Flags

Inheritance Flag	Compact Inheritance Flag	Description
`file_inherit`	`f`	Only inherit the ACL from the parent directory to the directory's files
`dir_inherit`	`d`	Only inherit the ACL from the parent directory to the directory's subdirectories
`inherit_only`	`I`	Inherit the ACL from the parent directory
`no_propagate`	`n`	Only inherit the ACL from the parent directory to the first-level contents of the directory
`-`	`n/a`	No permission granted
`successful_access`	`S`	Indicates whether an alarm or audit record should be initiated upon a successful access; used with audit or alarm ACE type
`failed_access`	`F`	Indicates whether an alarm or audit record should be initiated when an access fails; used with audit or alarm ACE types
`inherited`	`I`	Indicates that an ACE was inherited

24.5.1.4 ACL Properties

The ZFS file system includes the ACL properties to determine the specific behavior of ACL inheritance and ACL interaction with **chmod** command operations.

These properties are:

- aclinherit: Determine the behavior of ACL inheritance. Values include the following:

  discard: For new objects, no ACL entries are inherited when a file or directory is created. The ACL on the file or directory is equal to the permission mode of the file or directory.

  noallow: For new objects, only inheritable ACL entries that have an access type of deny are inherited.

  restricted: For new objects, the write _ owner and write _ acl permissions are removed when an ACL entry is inherited.

  passthrough: Files are created with a mode determined by the inheritable ACEs. If no inheritable ACEs exist that affect the mode, then the mode is set in accordance to the requested mode from the application.

  passthrough-x: Has the same semantics as passthrough except that when passthrough-x is enabled, files are created with the execute (x) permission only if the execute permission is set in file creation mode and in an inheritable ACE that affects the mode.

The default mode for ZFS aclinherit is restricted.

- aclmode: Modifies ACL behavior when a file is initially created or controls how an ACL is modified during a chmod operation. Values include the following:

  discard: Deletes all ACL entries that do not represent the mode of the file. This is the default value.

  mask: Reduces user or group permissions. The permissions are reduced such that they are no greater than the group permission bits unless it is a user entry that has the same UID as the owner of the file or directory. In this case, the ACL permissions are reduced so that they are no greater than owner permission bits. The mask value also preserves the ACL across mode changes, provided that an explicit ACL set operation has not been performed.

  passthrough: Indicates that no changes are made to the ACL other than generating the necessary ACL entries to represent the new mode of the file or directory.

The default mode for ZFS aclmode is discard.

24.5.2 Setting ACLs on ZFS Files

The basic rules of ACL access on a ZFS file are as follows:

1. ZFS processes ACL entries in the order they are listed in the ACL, from the top down.
2. Only ACL entries whose specified user matches the requester of the access are processed.
3. Once an allow permission has been granted, it cannot be denied by a subsequent ACL deny entry in the same ACL permission set.
4. The owner of the file is granted the `write _ acl` permission unconditionally even if the permission is explicitly denied. Otherwise, any permission left unspecified is denied. In the cases of deny permissions or when an access permission is missing, the privilege subsystem determines the access request that is granted for the owner of the file or for the superuser. This mechanism prevents owners of files from getting locked out of their files and enables the superuser to modify files for recovery purposes.

24.5.2.1 Command Syntax for Setting ACLs

To set or modify ACLs, use the `chmod` command. The command syntax resembles the syntax for setting permission bits on files, except that you specify A before typing the operator (+, =, or -).

Command syntax for trivial ACLs:

```
chmod [options] A[index]{+|=}owner@ |group@ |everyone@:\
access-permissions/...[:inheritance-flags]:deny | allow file

chmod [options] A-owner@, group@, everyone@:\
access-permissions/...[:inheritance-flags]:deny | allow file ...
chmod [options] A[index]- file
```

Command syntax for nontrivial ACLs:

```
chmod [options] A[index]{+|=}user|group:name:\
access-permissions/...[:inheritance-flags]:deny | allow file
chmod [options] A-user|group:name:\
access-permissions/...[:inheritance-flags]:deny | allow file ...
chmod [options] A[index]- file
```

The `chmod` command uses the following operators:

`A+` adds an ACL entry.

`A=` replaces an ACL entry.

To replace an entire ACL for a file, use this operator without specifying an index ID. In the following example, ACL entries for **file.1** are removed and replaced with the single entry for `everyone@`.

```
# chmod A=everyone@:read_data:allow file.0
```

`A-` removes an ACL entry.

To universally remove all nontrivial ACL entries for a file, use this operator and specify the file name without listing each entry to be removed. Use the following command syntax to restore a trivial ACL to the file.

```
# chmod A- filename
```

After you issue the command, only the entries for `owner@`, `group@`, and `everyone@` that comprise a trivial ACL remain.

For modifying existing ACLs: Using the operators without an index has a different effect from using them with an index. For example, `chmod  A=` replaces an entire ACL, while `chmod  A3=` replaces only the existing entry that has index number 3.

Permissions and inheritance flags are represented by unique letters listed in Table 24.2 and Table 24.4.

When you set ZFS ACLs, you can either use the letters that correspond to those permissions (compact mode) or type the permissions in full (verbose mode).

In the following example, both commands grant read and execute permissions to user **bob** on **file.1**:

```
# chmod A+user:bob:rx:allow file.0
# chmod A+user:bob:read_data/execute:allow file.0
```

To grant user **bob** inheritable read, write, and execute permissions for the newly created **d.2** and its files, you can use either one of the following commands:

```
# chmod A+user:bob:rwx:fd:allow d.2
# chmod A+user:bob:read_data/write_data/execute:file_inherit/
  dir_inherit: allow d.2
```

24.5.2.1.1 Displaying ACL Information

With the `ls` command, you can display ACL information in one of two formats: the `-v` option displays the permissions in full or verbose form; the `-V` option generates compact output by using letters that represent the permissions and flags.

The following examples show how the same ACL information is displayed in both verbose and compact format:

```
# ls -v file.0
-rw-r--r-- 1 root root 206695 Jul 20 14:27 file.0
0:owner@:read_data/write_data/append_data/read_attributes
/write_xattr/read_xattr/write_attributes/read_acl/write_acl
/write_owner/synchronize:allow
1:group@:read_data/read_attributes/read_xattr/read_acl
/synchronize:allow
```

```
2:everyone@:read_data/append_data/read_xattr/read_acl
/synchronize:allow
```

```
# ls -V file.0
-rw-r--r-- 1 root root 206695 Jul 20 14:27 file.0
owner@:rw-p--aARWcCos:-------:allow
group@:r-----a-R-c--s:-------:allow
everyone@:r-----a-R-c--s:-------:allow
```

For an explanation of the permissions for each user category, see Table 24.2.

24.5.2.2 Modifying ACLs on ZFS Files

This section provides several sample commands for setting and displaying ACLs. We encourage you to do the in-chapter exercises at the end of the following sections to familiarize yourself with these basic Solaris NFSv4 ACL methods. You must run the following examples as superuser, unless otherwise specified.

In the following example, read _ set permissions are given to user **bob** on a newly created file named **file.1**.

```
root@solaris_11_3:~# touch file.1
root@solaris_11_3:~# chmod A+user:bob:read_set:allow file.1
root@solaris_11_3:~# ls -V file.1
-rw-r--r--+ 1 root   root       0 Nov 4 11:58 file.1
           user:bob:r-----a-R-c---:-------:allow
           owner@:rw-p--aARWcCos:-------:allow
            group@:r-----a-R-c--s:-------:allow
       everyone@:r-----a-R-c--s:-------:allow
```

The next example gives user **mansoor** read and execute permissions on **file.1**.

```
root@solaris_11_3:~# chmod A+user:mansoor:rx:allow file.1
root@solaris_11_3:~# ls -V file.1
-rw-r--r--+ 1 root   root       0 Nov 4 11:58 file.1
    user:mansoor:r-x-----------:-------:allow
         user:bob:r-----a-R-c---:-------:allow
         owner@:rw-p--aARWcCos:-------:allow
          group@:r-----a-R-c--s:-------:allow
      everyone@:r-----a-R-c--s:-------:allow
```

In the next example, group@ permissions are changed to read and write for **file.1**.

```
root@solaris_11_3:~# chmod A3=group@:read_data/write_data:allow
                     file.1
root@solaris_11_3:~# ls -V file.1
-rw-rw-r--+ 1 root   root       0 Nov 4 11:58 file.1
    user:mansoor:r-x-----------:-------:allow
         user:bob:r-----a-R-c---:-------:allow
         owner@:rw-p--aARWcCos:-------:allow
         group@:rw------------:-------:allow
      everyone@:r-----a-R-c--s:-------:allow
```

In the following example, read _ data/execute permissions are added for the user **mansoor** on the **test.dir** directory.

```
root@solaris_11_3:~# mkdir test.dir
root@solaris_11_3:~# chmod A0+user:mansoor:rx:allow test.dir
root@solaris_11_3:~# ls -dV test.dir
drwxr-xr-x+ 2 root   root        2 Nov 4 12:07 test.dir
    user:mansoor:r-x-----------:-------:allow
         owner@:rwxp-DaARWcCos:-------:allow
         group@:r-x---a-R-c--s:-------:allow
      everyone@:r-x---a-R-c--s:-------:allow
```

In the following example, access permissions are removed for user **mansoor**.

```
root@solaris_11_3:~# chmod A0- test.dir
root@solaris_11_3:~# ls -dV test.dir
drwxr-xr-x 2 root   root        2 Nov 4 12:07 test.dir
         owner@:rwxp-DaARWcCos:-------:allow
          group@:r-x---a-R-c--s:-------:allow
         everyone@:r-x---a-R-c--s:-------:allow
```

24.5.2.3 ACL Interaction with Permission Bits

In ZFS files, the UNIX permission bits correspond to the ACL entries, but are stored in a special cache. When you change a file's permission bits, the file's ACL is updated accordingly. Similarly, modifying a file's ACLs causes changes in the permission bits.

For more information about permission bits, see the man page for chmod.

It would be advisable at this point to review the setting of UNIX permission bits in octal format, presented in Chapter 5, Sections 4 and 5.

The following examples show the relationship between a file or directory's ACLs and the permission bits, and illustrate how permission changes in one affect the other and vice versa.

The first example begins with the following ACL for **file.2**, whose permission bits are set to 644.

```
root@solaris_11_3:~# touch file.2
root@solaris_11_3:~# ls -V file.2
-rw-r--r-- 1 root    root      0 Nov 4 12:09 file.2
          owner@:rw-p--aARWcCos:-------:allow
           group@:r-----a-R-c--s:-------:allow
       everyone@:r-----a-R-c--s:-------:allow
```

The following chmod command removes the ACL entry for everyone@ on **file.2**. The read permission bits for everyone@ are also removed and are changed to 640.

```
root@solaris_11_3:~# chmod A2- file.2
root@solaris_11_3:~# ls -V file.2
-rw-r-----+ 1 root   root       0 Nov 4 12:09 file.2
           owner@:rw-p--aARWcCos:-------:allow
           group@:r-----a-R-c--s:-------:allow
```

In the next example, the ACL is replaced with just read _ data/write _ data permissions for everyone@ on **file.2**. No owner@ or group@ ACL entry exists to override the permissions for owner and group. So the permission bits become 666.

```
root@solaris_11_3:~# chmod A=everyone@:rw::allow file.2
root@solaris_11_3:~# ls -V file.2
-rw-rw-rw-+ 1 root   root        0 Nov 4 12:09 file.2
        everyone@:rw------------:-------:allow
```

Notice that all other ACL entries have been removed.

In the next example, the ACL is replaced with read permissions just for user **mansoor**. The command, however, leaves no trivial ACL entries. Consequently, the permission bits are set to 000, which denies **mansoor** access to **file.2**. The file effectively becomes inaccessible.

```
root@solaris_11_3:~# chmod A=user:mansoor:r::allow file.2
root@solaris_11_3:~# ls -V file.2
----------+ 1 root root 0 Nov 4 12:09 file.2
         user:mansoor:r-------------:-------:allow
```

Notice in the ls –V output that no trivial ACL entries exist.

The next example shows how setting permission bits also updates the ACL. The bits for **file.2** are set to 655. Automatically, default trivial ACL permissions are set.

```
root@solaris_11_3:~# chmod 655 file.2
root@solaris_11_3:~# ls -V file.2
-rw-r-xr-x 1 root    root      0 Nov 4 12:09 file.2
          owner@:--x-----------:-------:deny
          owner@:rw-p--aARWcCos:-------:allow
           group@:r-x---a-R-c--s:-------:allow
      everyone@:r-x---a-R-c--s:-------:allow
```

The following examples illustrate how specific aclmode and aclinherit property values affect ACL behavior. If these properties are set, ACL permissions for a file or directory are either reduced or expanded to be consistent with the associated group.

Suppose that the aclmode property is set to mask and the aclinherit property is set to restricted in the pool, and that the original file and group ownership and ACL permissions are as follows:

```
root@solaris_11_3:~# zfs set aclmode=mask rpool/export/home/bob
root@solaris_11_3:~# zfs set aclinherit=restricted rpool/export/
                     home/bob
```

```
root@solaris_11_3:~# ls -lV file.1
-rw-rw-r--+ 1 root    root      0 Nov 4 11:58 file.1
      user:mansoor:r-x-----------:-------:allow
        user:bob:r-----a-R-c---:-------:allow
         owner@:rw-p--aARWcCos:-------:allow
         group@:rw------------:-------:allow
       everyone@:r-----a-R-c--s:-------:allow
```

A chown operation changes **file.1**'s ownership to **bob** and the group **staff**.

```
root@solaris_11_3:~# chown bob:staff file.1
```

Bob then changes **file.1**'s permission bits to 640. Because of the ACL properties that were previously set, the permissions for the groups in the ACL are reduced in order to not exceed the permissions of the owning group **staff**. Please notice how **bob** drops out of the superuser role to accomplish the following:

```
root@solaris_11_3:~# su - bob
Oracle Corporation      SunOS 5.11  11.3  September 2015
bob@solaris_11_3:~$ ls -lV file.1
-rw-rw-r--+ 1 bob staff 0 Nov 4 11:58 file.1
    user:mansoor:r-x-----------:-------:allow
         user:bob:r-----a-R-c---:-------:allow
         owner@:rw-p--aARWcCos:-------:allow
         group@:rw------------:-------:allow
     everyone@:r-----a-R-c--s:-------:allow
bob@solaris_11_3:~$ chmod 640 file.1

bob@solaris_11_3:~$ ls -lV file.1
-rw-r-----+ 1 bob staff 0 Nov 4 11:58 file.1
    user:mansoor:r-------------:-------:allow
         user:bob:r-----a-R-c---:-------:allow
         owner@:rw-p--aARWcCos:-------:allow
         group@:r-----a-R-c--s:-------:allow
     everyone@:------a-R-c--s:-------:allow
```

Bob then changes the permission bits to 770. Consequently, the permissions of the groups in the ACL are also changed to match the permission of the owning group **staff**.

```
bob@solaris_11_3:~$ chmod 770 file.1
bob@solaris_11_3:~$ ls -lV file.1
-rwxrwx---+ 1 bob staff 0 Nov 4 11:58 file.1
      user:mansoor:r-x-----------:-------:allow
        user:bob:r-----a-R-c---:-------:allow
        owner@:rwxp--aARWcCos:-------:allow
        group@:rwxp--aARWc--s:-------:allow
      everyone@:------a-R-c--s:-------:allow
```

24.5.3 Setting ACL Inheritance on ZFS Files

You can determine how ACLs are inherited on files and directories. The `aclinherit` property can be set globally on a file system. By default, `aclinherit` is set to restricted.

24.5.3.1 Granting ACLs That Are Inherited by Files

This section identifies the file ACEs that are applied when the `file _ inherit` flag is set.

In the following example, `read _ data/write _ data` permissions are added for files in the **test2.dir** directory for user **mansoor** so that he has read access on any newly created files. Also notice the switch back to superuser.

```
bob@solaris_11_3:~$ su
Password:xxx
root@solaris_11_3:~# mkdir test2.dir
root@solaris_11_3:~# chmod A+user:mansoor:rw:f:allow test2.dir
root@solaris_11_3:~# ls -dV test2.dir
drwxr-xr-x+ 2 root    root      2 Nov 4 12:23 test2.dir
     user:mansoor:rw------------:f------:allow
          owner@:rwxp-DaARWcCos:-------:allow
          group@:r-x---a-R-c--s:-------:allow
      everyone@:r-x---a-R-c--s:-------:allow
```

In the following example, user **mansoor**'s permissions are applied on the newly created **test2.dir/file.2** file. The ACL inheritance granted, `read _ data:file _ inherit:allow`, means user **mansoor** can read the contents of any newly created file.

```
root@solaris_11_3:~# touch test2.dir/file.2
root@solaris_11_3:~# ls -V test2.dir/file.2
-rw-r--r--+ 1 root    root      0 Nov 4 12:26 test2.dir/file.2
     user:mansoor:r-------------:------I:allow
          owner@:rw-p--aARWcCos:-------:allow
          group@:r-----a-R-c--s:-------:allow
      everyone@:r-----a-R-c--s:-------:allow
```

Because the `aclinherit` property for this file system is set to the default mode, `restricted`, user **mansoor** does not have `write _ data` permission on **file.2** because the group permission of the file does not allow it.

The `inherit _ only` permission, which is applied when the `file _ inherit` or `dir _ inherit` flags are set, is used to propagate the ACL through the directory structure. As such, user **mansoor** is granted or denied permission from `everyone@` permissions only if he is the file owner or is a member of the file's group owner. For example:

```
root@solaris_11_3:~# mkdir test2.dir/subdir.2
root@solaris_11_3:~# ls -dV test2.dir/subdir.2
drwxr-xr-x+ 2 root    root      2 Nov 4 12:27 test2.dir/subdir.2
     user:mansoor:rw------------:f-i---I:allow
          owner@:rwxp-DaARWcCos:-------:allow
          group@:r-x---a-R-c--s:-------:allow
       everyone@:r-x---a-R-c--s:-------:allow
```

24.5.3.2 Granting ACLs That Are Inherited by Both Files and Directories

This section provides examples that identify the file and directory ACLs that are applied when both the `file _ inherit` and `dir _ inherit` flags are set.

In the following example, user **mansoor** is granted read, write, and execute permissions that are inherited for newly created files and directories.

```
root@solaris_11_3:~# mkdir test3.dir
root@solaris_11_3:~# chmod A+user:mansoor:rwx:fd:allow test3.dir
root@solaris_11_3:~# ls -dV test3.dir
drwxr-xr-x+ 2 root    root      2 Nov 4 12:29 test3.dir
     user:mansoor:rwx-----------:fd-----:allow
         owner@:rwxp-DaARWcCos:-------:allow
         group@:r-x---a-R-c--s:-------:allow
      everyone@:r-x---a-R-c--s:-------:allow
```

The inherited text in the following output is an informational message that indicates that the ACE is inherited.

```
root@solaris_11_3:~# touch test3.dir/file.3
root@solaris_11_3:~# ls -V test3.dir/file.3
-rw-r--r--+ 1 root    root      0 Nov 4 12:31 test3.dir/file.3
     user:mansoor:r-------------:------I:allow
         owner@:rw-p--aARWcCos:-------:allow
         group@:r-----a-R-c--s:-------:allow
      everyone@:r-----a-R-c--s:-------:allow
```

In these examples, because the permission bits of the parent directory for `group@` and `everyone@` deny write and execute permissions, user **mansoor** is denied write and execute permissions. The default `aclinherit` property is `restricted`, which means that `write _ data` and execute permissions are not inherited.

In the following example, user **mansoor** is granted read, write, and execute permissions that are inherited for newly created files but are not propagated to subsequent contents of the directory.

```
root@solaris_11_3:~# mkdir test4.dir
root@solaris_11_3:~# chmod A+user:mansoor:rwx:fn:allow test4.dir
root@solaris_11_3:~# ls -dV test4.dir
drwxr-xr-x+ 2 root    root      2 Nov 4 12:33 test4.dir
     user:mansoor:rwx-----------:f--n---:allow
         owner@:rwxp-DaARWcCos:-------:allow
         group@:r-x---a-R-c--s:-------:allow
      everyone@:r-x---a-R-c--s:-------:allow
```

As the following example illustrates, **mansoor's** read _ data/write _ data/execute permissions are reduced based on the owning group's permissions.

```
root@solaris_11_3:~# touch test4.dir/file.4
root@solaris_11_3:~# ls -V test4.dir/file.4
-rw-r--r--+ 1 root    root      0 Nov 4 12:35 test4.dir/file.4
     user:mansoor:r-------------:------I:allow
         owner@:rw-p--aARWcCos:-------:allow
         group@:r-----a-R-c--s:-------:allow
      everyone@:r-----a-R-c--s:-------:allow
```

24.5.3.3 Modifying ACL Inheritance with the ACL Inherit Mode

This section describes the aclinherit property values.

ACL Inheritance with the ACL Inherit Mode Set to discard:

If the aclinherit property on a file system is set to discard, then ACLs can potentially be discarded when the permission bits on a directory change. For example:

```
root@solaris_11_3:~# zfs set aclinherit=discard rpool/export/home/
                     bob
root@solaris_11_3:~# mkdir test5.dir
root@solaris_11_3:~# chmod A+user:mansoor:rwx:d:allow test5.dir
root@solaris_11_3:~# ls -dV test5.dir
drwxr-xr-x+ 2 root    root      2 Nov 4 12:37 test5.dir
     user:mansoor:rwx-----------:-d-----:allow
         owner@:rwxp-DaARWcCos:-------:allow
         group@:r-x---a-R-c--s:-------:allow
      everyone@:r-x---a-R-c--s:-------:allow
```

If, at a later time, you decide to tighten the permission bits on a directory, the nontrivial ACL is discarded. For example:

```
root@solaris_11_3:~# chmod 744 test5.dir
root@solaris_11_3:~# ls -dV test5.dir
drwxr--r--+ 2 root    root      2 Nov 4 12:37 test5.dir
     user:mansoor:r-------------:-d-----:allow
         owner@:rwxp-DaARWcCos:-------:allow
         group@:r-----a-R-c--s:-------:allow
      everyone@:r-----a-R-c--s:-------:allow
```

ACL Inheritance with the ACL Inherit Mode Set to noallow

In the following example, two nontrivial ACLs with file inheritance are set. One ACL allows read _ data permission, and one ACL denies read _ data permission. This example also illustrates how you can specify two ACEs in the same chmod command.

```
root@solaris_11_3:~# zfs set aclinherit=noallow rpool/export/home/
                     bob
```

```
root@solaris_11_3:~# mkdir test6.dir
root@solaris_11_3:~# chmod A+user:mansoor:r:f:deny,user:bob:r:f:
                     allow test6.dir
root@solaris_11_3:~# ls -dV test6.dir
drwxr-xr-x+ 2 root    root      2 Nov 4 12:45 test6.dir
     user:mansoor:r-------------:f------:deny
         user:bob:r-------------:f------:allow
         owner@:rwxp-DaARWcCos:-------:allow
         group@:r-x---a-R-c--s:-------:allow
      everyone@:r-x---a-R-c--s:-------:allow
```

As the following example shows, when a new file is created, the ACL that allows read _ data permission is discarded.

```
root@solaris_11_3:~# touch test6.dir/file.6
root@solaris_11_3:~# ls -V test6.dir/file.6
-rw-r--r--+ 1 root    root      0 Nov 4 12:48 test6.dir/file.6
     user:mansoor:r-------------:------I:deny
         owner@:rw-p--aARWcCos:-------:allow
         group@:r-----a-R-c--s:-------:allow
      everyone@:r-----a-R-c--s:-------:allow
```

EXERCISE 24.14

Execute all of the command line example code in Section 24.5.2.2 and its subsections on your Solaris system, and verify that they give the same output as shown.

EXERCISE 24.15

Execute all of the command line example code shown in Section 24.5.3 and its subsections, and verify that they give the same output as shown.

SUMMARY

This chapter provided a common user of PC-BSD or Solaris with the basic techniques of working with the Zettabyte File System (ZFS).

We first went over some basic terms used in ZFS and then described what ZFS is from the user perspective. What differentiates ZFS from other file systems is that ZFS file systems are mapped onto pool storage facilities, known as zpools, rather than onto physical storage media like disk drives. The zpools are then mapped onto physical media. That means that the file system's storage requirements can grow as more physical media devices are added to the zpools.

We then provided six working examples of using the two ZFS commands, zpool and zfs. These working examples illustrate for the beginner some of the basic operations that can be used to create pools and file systems.

We provided a command reference section that gives many ZFS command usage examples.

We provided file system backup procedures and a Bourne shell script example that used the `zfs snapshot`, `zfs send`, and `zfs receive` commands.

We finally illustrated how to manage ACLs on ZFS files and directories for Solaris, with several examples.

QUESTIONS AND PROBLEMS

1. Is it possible to create a zpool using only a single slice on a vdev? Does your answer reflect the fact that only one file system can exist on that slice?
2. List the advantages and disadvantages that ZFS has for you on your UNIX system.
3. Give a brief description of the `zdb` command.
4. Similar to Example 24.3, create a mirrored zpool using two files that simulate disk drives, and are 256 MB in size each. Name the files **disk1** and **disk2**. Then answer the following questions:
   a. What is the pathname to any file you can create in the zpool?
   b. If you create a 32 MB file in your zpool, what size increase do you see in the files **disk1** and **disk2**?
   c. How much free space is now in the zpool?
5. Define the following terms in ZFS:
   Scrubbing, *resilvering*, *slicing*, *mirroring*, *zpool*, *vdev*.
6. Does PC-BSD allow you to create a mirrored zpool across two disks at the initial installation of the system? Does Solaris?
7. If you create a zpool named **pool1**, and a file system on that zpool named **bobsfiles** with the `zfs` command, what is the pathname to a file named **data27** in that file system? What is the exact syntax of the command you used to create the file system?
8. List eight of the basic `zfs` subcommands that allow you to do file and file system backups and archiving on Solaris and PC-BSD.
9. Following the completion of the first five steps of Example 24.5, do the following:
   a. Use the `zfs` command to create a dataset named **usbdrive** on pool **backup** located on the thumb drive inserted into your Solaris system. The name of the dataset would be **backup/usbdrive**.
   b. Type the command `zfs set copies=2 backup/usbdrive`.
   c. What you have achieved with this command is a signature validation of using ZFS and creating a zpool on the USB thumb drive. The USB thumb drive is a redundant device to the extent shown. But more importantly, by setting the property of `copies=2` on this dataset, you have made the USB thumb drive redundant to itself, because ZFS now keeps two copies of everything you put in the dataset **backup/usbdrive**. And you can use ZFS facilities to ensure integrity of the data to the bit level on the USB thumb drive. Given how inexpensive USB thumb drives

are, even in larger capacities, having two automatically created copies of your files on this thumb drive is not prohibitive.

d. Copy a number of important files into this new dataset from your systems hard drive using either the `cp` or `rsync` commands shown in Chapter 23.
e. Instead of proceeding onto step 6 of the example, retain the zpool and dataset you have created on the thumb drive, and use it as a backup drive for your important files. You can periodically use `rsync` to keep the backup files synchronized to the important files on your hard drive. You may even decide that the important files you want to back up to the thumb drive are in a single directory or multiple directories. You can then use `rsync` to copy directories over to the thumb drive.
f. To remove the USB thumb drive temporarily at any time, use the `zfs unmount` command. Then you can remove it from the computer. Remember to use the `zfs mount` command when you want to reinsert the USB thumb drive and archive or backup files to it.

10. Repeat Problem 24.9 on your Solaris system, but instead of using `cp` or `rsync` to move important files to the USB thumb drive from some arbitrary place(s), do the following:
    a. Create a new zpool on the hard drive which will be the source of your important files to be backed up.
    b. Put a dataset in that new pool and fill the dataset with important files you want to backup.
    c. Use the methods of Example 24.2 to create a ZFS snapshot of the new dataset created on your hard drive pool. Then, continuing the methods shown in Example 24.2, use `zfs send` and `zfs receive` to copy the snapshot from the hard drive to the USB thumb drive.

11. On a PC-BSD system, back up your home directory (or a selected subdirectory of your home directory) to a USB thumb drive using the commands `zfs snapshot`, `zfs send`, and `zfs receive`. The ZFS dataset for your home directory should be named something like **tank/usr/home/your_username**.

    Be sure to insert and test the USB thumb drive you want to use for this problem, to find out if it is usable on your system. If it doesn't show up in **/dev** after insertion, it is not usable!

    After testing the USB thumb drive for usability, make sure that automounting of removable media is turned off in System Settings. Prepare the USB thumb drive using the Disk Manager to destroy any partition table on it, and create a GPT partitioning table on it. Then use the `gpart` command to create one primary partition with a ZFS file system.

12. On a PC-BSD system, use an externally mounted USB hard drive to repeat the operations given in Example 24.6. Note any differences in using a hard disk on the USB bus and using a USB thumb drive.

13. An important use of the procedure that you will implement in this problem is to enable you to create a ZFS mirror of your system disk, so that you have redundancy of your system disk and data. This single, internally mounted hard disk vdev, which can also contain all of your user data files, may be inside of a laptop computer that can only have a single hard disk internally mounted. The mirror you will create in this problem is maintained over the USB bus. If the hard drive in the external (but openable) enclosure we designate here is a SATA drive, then that drive can also be used as a replacement for the internal, single SATA hard disk drive if that drive fails. The problem assumes that the external USB hard disk in its enclosure, and the internal hard disk in your computer, can be easily removed and reinstalled. They must also both be SATA drives, either spinning disk or SSD. This problem achieves the same objectives as those illustrated in Chapter 23 for Clonezilla Live.

    On a PC-BSD system, use an externally mounted USB hard drive in an openable enclosure to repeat the operations given in Example 24.4. You must make sure that the capacity of this USB external drive is large enough to accommodate the creation of the mirror.

    To test whether or not you have correctly solved this problem, shut down the system. Then remove the external USB hard drive from its enclosure, and replace the internally mounted hard drive in your computer with it. Your system should then boot from the new hard drive, but the mirror will show as being in a degraded state. Then replace original and external disks in the original configuration.

    To repeat this procedure when your internal hard drive actually fails, follow the processes shown previously in the chapter for replacements of vdevs in a mirror.

14. Mansoor is working with Bob on a project. He needs to be able to read, write, create, and delete files related to the project, which are located in the **Project** directory of Bob's home directory. Bob and Mansoor are ordinary users without administrative privileges. They wish to do this project without contacting the system administrator to request new groups, group membership changes, `sudo` changes, and so on.

    When the project is over, Bob will remove the modify permissions on his home and the **Project** directory for user **mansoor**. By modify, we mean Mansoor can still look at the contents of the **Project** directory, but cannot make any changes to the files in it. He will do this himself, instead of contacting the system administrator.

    On your own Solaris system, in conjunction with another user, use ACLs to accomplish the following (substituting valid usernames on your system for Bob and Mansoor):

    a. Create a project directory under Bob's home directory named **Project**. On our system, the path to this new directory is **`/export/home/bob/Project`**.
    b. Set the ACL on Bob's home directory so Mansoor has read, write, and execute privileges on it.
    c. Set the ACL on the **Project** directory so that Mansoor has `rwxo` privileges on it.
    d. Have Bob create some files in the **Project** directory.
    e. Have Mansoor make Bob's home directory the current directory.

f. Have Mansoor test whether or not he can:
   - Delete files in Bob's home directory
   - Delete the **Project** directory from Bob's home directory
   - List, create new files, or remove the files that Bob put in the **Project** directory

g. Have Bob revoke Mansoor's modify privileges on Bob's home directory and the **Project** directory.

h. Have Mansoor test the revocation of modify privileges from step g.

i. Why can Mansoor still see the files in Bob's home directory, and the files in the **Project** directory, but not delete or modify any files in those directories after step g?

j. What `chmod` command(s) would Bob have to execute to deny Mansoor access to his home directory?
   Show verification of ACL settings at as many steps as necessary to validate what you have done.

15. If you give a set of users permissions to a project directory using ACLs, how can you ensure that subdirectories that are created by the project manager beneath that project directory provide the same access privileges to those users?

16. Create a project directory on your system and create a Git repository in it for any number of local users on your computer system. Then use ACLs to give access to the project directory to the users that are collaborating in the project. This should allow those users to push to and pull from the Git repository. Have your allowed users test the repository. Also test the security of the repository; that is, can nonallowed users access it?
    See Chapter 17, Section 5.7 for more information on creating a Git repository.

CHAPTER 25

Virtualization Methodologies

Objectives

- To give background information on operating system virtualization
- To provide a description of PC-BSD `iocage` jails virtualization method
- To illustrate `iocage` basic usage
- To give command references for `iocage` networking, and jail types
- To list `iocage` best practices, and provide examples of advanced usage
- To explicitly detail `iocage` installation on PC-BSD
- To give a set of complete worked examples of `iocage` usage
- To describe Solaris zones virtualization method
- To define and illustrate the nonglobal zone state model
- To show Solaris commands that affect the zone state
- To show how to create a Solaris Zone
- To provide an example of installing a web server in a Solaris zone
- To describe and illustrate VirtualBox with different types of installation examples
- How to install VirtualBox on PC-BSD
- How to install VirtualBox on Solaris
- How to install Solaris as a guest on a PC-BSD host
- To show how to secure an FTP server in a VirtualBox guest
- How to install PC-BSD as a guest on a LINUX or Windows host

- How to install Solaris as a guest on a LINUX or Windows host
- To cover the commands and primitives (Solaris-only commands in parentheses)

```
dladm (Solaris), ipadm(Solaris), ifconfig, jail, pkg,
pkgadd(Solaris), VBoxManage, VirtualBox, iocage, xz ,
zoneadm(Solaris), zonecfg(Solaris), zonename(Solaris)
```

25.1 INTRODUCTION TO VIRTUALIZATION METHODOLOGIES AND BACKGROUND

Computer hardware virtualization is the simulation, to various degrees, of hardware platforms, parts of them, or only the functionality required to run one or more operating systems (OSs). It abstracts and thus hides the physical characteristics of the hardware from the users. Traditionally, the software that controlled virtualized machines was known as the *hypervisor*. Currently, the hypervisor is often called a *virtual machine monitor* (VMM). For example, in VirtualBox, the *virtual machine* (VM) VirtualBox manager is the graphical front-end for a VMM.

Platform virtualization is accomplished on any given hardware by host software (the hypervisor), which creates a simulated computer environment, a VM, for its guest software. The guest software can be as small as a single user application or as large as complete OSs. The guest software executes as if it were running directly on the physical hardware.

Virtualization comes with some performance disadvantages, both in resources required to run the hypervisor and in reduced performance on the VM guest compared with running applications on a nonvirtualized host physical machine.

A VM can be more easily controlled and inspected from outside than a physical one, and its configuration is more flexible. This is very useful in kernel development and for teaching OS courses. A new VM can be implemented as needed without the need for an up-front hardware purchase.

A VM can easily be moved from one physical machine to another as needed. An unrecoverable fault inside a VM guest does not harm the host system, so there is no risk of crashing the host OS.

Examples of virtualization implementations:

1. Running one or more applications that are not supported by the host OS: A VM running the required guest OS could allow the desired applications to be run, without altering the host OS.
2. Evaluating an alternate OS: The new OS could be run within a VM, without altering the host OS.
3. Server virtualization: Multiple virtual servers in containers could be run on a single physical server, to utilize more fully the hardware resources of the physical server.
4. Duplicating specific environments: A VM could, depending on the virtualization software used, be duplicated and installed on multiple hosts, or restored to a previously backed-up system state.

5. Creating a protected environment: If a guest OS running on a VM becomes damaged in a way that is difficult to repair, such as may occur when testing, the VM can be discarded without harm to the host system, and a clean copy used next time.

Primary contemporary virtualization techniques:

Full virtualization: In full virtualization, the VM simulates enough hardware to allow a complete "guest" OS, one designed for the same processor instruction set architecture (ISA) to be run in isolation.

Examples for UNIX systems running on X86 ISAs include VirtualBox, Parallels Workstation, Oracle VM, Virtual Server, Hyper-V, VMware Workstation, and VMware.

Hardware-assisted virtualization: In hardware-assisted virtualization, the hardware provides architectural support that facilitates building a VMM and allows guest OSs to be run in isolation. In 2005 and 2006, Intel and AMD provided additional hardware to support virtualization. Sun Microsystems added similar features in their UltraSPARC processors in 2005. Examples of virtualization platforms adapted to such hardware include KVM, VMware Workstation, VMware Fusion, Hyper-V, Xen, Oracle VM server for SPARC, and VirtualBox.

Partial virtualization: In partial virtualization, including address space virtualization, the VM simulates multiple instances of much of an underlying hardware environment, particularly address spaces.

Paravirtualization: In paravirtualization, the VM does not necessarily simulate hardware, but instead (or in addition) offers a special application programmer's interface (API) that can only be used by modifying the "guest" OS. For this to be possible, the "guest" OS's source code must be available.

OS-level virtualization: In OS-level virtualization, a physical server is virtualized at the OS level, enabling multiple isolated and secure virtualized servers to run on a single physical server. The "guest" OS environments share the same running instance of the OS as the host system. Thus, the same OS kernel is also used to implement the "guest" environments, and applications running in a given "guest" environment view it as a stand-alone system. Examples in UNIX are FreeBSD jails, `iocage` in PC-BSD, and other examples include Solaris zones. In LINUX, examples include LXC, and its derivative management system Docker.

In this chapter, we illustrate both full virtualization and OS-level virtualization. We use three popular and important facilities for creating a virtual environment within which a UNIX OS can work. What differentiates these three facilities is that, for the first two, all virtual environments are running under the same kernel (OS-level virtualization). In the third one, any number of different kernels can be running simultaneously on one machine (full virtualization).

Practically speaking, an important application of these implementations, as already stated, is to provide a measure of system security, in addition to what is described in

Section 23.9. But that is not the only reason an ordinary user or system administrator would deploy the virtualization methods we demonstrate.

A user might want to take advantage of some of the facilities that an additional OS offers, above and beyond what is available as the main OS into which the computer boots. Instead of shutting down the main OS, and then booting into the additional OS, both can be run simultaneously using the virtualization method shown here. For example, Solaris does not offer LibreOffice as an application, but PC-BSD does by default. So if you run both OSs simultaneously, you can use LibreOffice and Solaris applications and facilities on the same machine. Of course, there are trade-offs in doing this, mainly in terms of performance speed and disk usage.

Also advantageous is the deployment of VMs to allow you to "test drive" a particular OS without devoting an entire hardware platform to it. This can also be achieved by running a "live" version of it from a DVD or a USB thumb drive, but the performance speed and persistence of data using these techniques is somewhat limited. In this chapter, we show several examples of installing guest VMs on mainstream OSs, such as LINUX and Windows, to allow you to "test drive" our base UNIX systems in a more fully functional way.

25.1.1 Virtualized Network Addresses in PC-BSD and Solaris

In the examples shown in the next sections, it is necessary to have the VM guest system network connection "bridged" to the host system. We show how to do this in Example 25.8, but as a preliminary consideration, you need to be able to quickly discover what the network address of a PC-BSD or Solaris machine is. That is true whether you are running either of those systems as a stand-alone, or inside of a VM environment. We also detail the commands we use here in Section 23.8.3, "Network Configuration."

For Solaris, you use the `ipadm show-addr` command, and on PC-BSD you use the `ifconfig` command, as follows:

On Solaris:
```
bob@solaris113beta:~$ dladm show-link
LINK                CLASS     MTU    STATE    OVER
net0                phys      1500   up       --
bob@solaris113beta:~$ ipadm show-addr
ADDROBJ            TYPE     STATE       ADDR
lo0/v4             static   ok          127.0.0.1/8
net0/v4            dhcp     ok          192.168.0.15/24
lo0/v6             static   ok          ::1/128
net0/v6            addrconf ok          fe80::a00:27ff:febb:de8e/10
bob@solaris113beta:~$
```

The Internet Protocol (IP) address of this Solaris machine, as shown in bold type in the `ipadm` command output, is 192.168.0.15.

On PC-BSD:
```
[bob@pcbsd-1233] ~% ifconfig
```

```
em0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0
mtu1500options=9b<RXCSUM,TXCSUM,VLAN_MTU,VLAN_HWTAGGING,VLAN_
HWCSUM>
        ether 08:00:27:38:ba:74
        inet6 fe80::a00:27ff:fe38:ba74%em0 prefixlen 64 scopeid 0x1
        inet 192.168.0.14 netmask 0xffffff00 broadcast
        192.168.0.255
        nd6 options=23<PERFORMNUD,ACCEPT_RTADV,AUTO_LINKLOCAL>
        media: Ethernet autoselect (1000baseT <full-duplex>)
        status: active
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> metric 0 mtu 16384
        options=600003<RXCSUM,TXCSUM,RXCSUM_IPV6,TXCSUM_IPV6>
        inet6 ::1 prefixlen 128
        inet6 fe80::1%lo0 prefixlen 64 scopeid 0x2
        inet 127.0.0.1 netmask 0xff000000
        nd6 options=21<PERFORMNUD,AUTO_LINKLOCAL>
[bob@pcbsd-1233] ~%
```

The IP address of this PC-BSD machine, as shown in bold type on the command output, is 192.168.0.14.

25.2 PC-BSD JAILS WITH iocage

A jail is an implementation of OS-level virtualization that effectively isolates a process tree from the rest of the system. Its historic precursor is the `chroot` command, which allows a privileged user to change the root of the file system for a process to a specifically isolated location, so that other processes and users are insulated from interacting with the "chrooted" process. Previously, in PC-BSD, jails were managed by the Warden jail management system.

Jails are very similar to Linux containers, which are used in such management systems as LXC and Docker. The major difference between jails in PC-BSD and containers in Linux is that Linux has some very specialized system calls that allow namespace operations and isolation in several domains, and the establishment of cgroups.

In the following sections, we describe the facilities of `iocage`, a modern PC-BSD jail management program. We also give a number of examples of how to work with jails using `iocage`. We conclude with worked examples of `iocage` installation and jail creation and management.

25.2.1 iocage Introduction, Overview, and Use

`iocage` is a jail/container manager that uses some of the best features and technologies the PC-BSD OS has to offer, particularly ZFS. It is easy to use, and has a simple and easy-to-understand command syntax, as follows:

```
iocage sub-command [-option(s)] [command argument(s)]
[properties],
```

where the command argument(s) are usually UUID or TAG, which are the unique identifier or name of a jail.

Listing:

```
iocage activate ZPOOL
iocage cap UUID|TAG
iocage chroot UUID|TAG [command]
iocage clean [-a|-b|-j]
iocage clone UUID|TAG [UUID|TAG@snapshot] [property=value]
iocage console UUID|TAG
iocage create [-e] [base=[RELEASE|NAME]] [pkglist=file]
[property=value]
iocage deactivate ZPOOL
iocage defaults
iocage destroy [-f] UUID|TAG
iocage df
iocage exec [-u username | -U username] UUID|TAG|ALL command [arg
...]
iocage export UUID|TAG
iocage fetch [release=RELEASE | ftphost=ftp.hostname.org |
ftpdir=/dir/ |
              ftpfiles="base.txz doc.txz lib32.txz src.txz"]
iocage get property|all UUID|TAG
iocage help
iocage import UUID [property=value]
iocage init-host IP ZPOOL
iocage inuse UUID|TAG
iocage limits [UUID|TAG]
iocage list [-b|-t|-r]
iocage package UUID|TAG
iocage promote UUID|TAG
iocage rcboot
iocage reboot UUID|TAG
iocage rcshutdown
iocage record start|stop UUID|TAG
iocage reset UUID|TAG|ALL
iocage restart UUID|TAG
iocage rollback UUID|TAG@snapshotname
iocage runtime UUID|TAG
iocage set property=value UUID|TAG
iocage show property
iocage snaplist UUID|TAG
iocage snapremove UUID|TAG@snapshotname|ALL
iocage snapshot [-r] UUID|TAG [UUID|TAG@snapshotname]
iocage start UUID|TAG
iocage stop UUID|TAG|ALL
iocage uncap UUID|TAG
iocage update UUID|TAG
iocage upgrade UUID|TAG [release=RELEASE]
iocage version | --version
```

For more information, read the manual page for `iocage`. This extensive manual page gives you all of the options, subcommands, and properties that you can use with the `iocage` command.

25.2.2 Basic Usage

This section details basic `iocage` command usage.

All of the commands are executed as superuser.

Fetch a release: The first step with `iocage` on a "pristine" system (where it has never been run before) is to fetch a release. By default, `iocage` will attempt to fetch the host's current release from the freebsd.org servers. Once the release is downloaded, the most recent patches are applied.

```
# iocage fetch
```

If a specific release is required, run the following. In Section 25.2.9, we fetched Release 10.2 on our PC-BSD system. Be aware that a later release of `iocage` may be available to you.

```
# iocage fetch release=10.2-RELEASE
```

In the case that a specific download mirror is required:

```
# iocage fetch ftphost=ftp.hostname.org
```

You can also specify an FTP directory from which to fetch the base files:

```
# iocage fetch ftpdir=/dir/
```

Creating a jail: There are three supported basic jail types: full, clone, and base jail. In addition to these three, there are two more (empty and templates). Depending on requirements, the `create` subcommand can be applied to create any of the three types. By default, `iocage` will create a fully independent jail of the current host's release and set the `TAG` property to today's date.

For example:

```
# iocage create
```

This will create a fully independent jail.

To create a lightweight jail (clone):

```
# iocage create -c
```

To create a base jail:

```
# iocage create -b
```

To create a lightweight jail and set its IP address and `tag` name to `myjail`:

```
# iocage create -c tag=myjail ip4_addr="em0|10.1.1.10/24"
```

Listing jails: To list all jails:

```
# iocage list
```

To see all downloaded releases:

```
# iocage list -r
```

To see available templates:

```
# iocage list -t
```

Starting, stopping, or restarting a jail: To start or stop any jail on the system, both `UUID` and `TAG` can be used interchangeably. To simplify `UUID` handling, `iocage` also accepts a partial `UUID` with any subcommand.

To start a jail tagged `myjail`:

```
# iocage start myjail
```

To start a jail with a full `UUID` run:

```
# iocage start 26e8e027-f00c-11e4-8f7f-3c970e80eb61
```

Or, to start the jail only with a partial `UUID`, enter the first few characters only:

```
# iocage start 26e8
```

To stop a jail, just use the `stop` subcommand instead of `start`:

```
# iocage stop myjail
```

To restart a jail run, which must already be running:

```
# iocage restart myjail
```

Note: Short `UUID`s are supported with all operations and subcommands within `iocage`.

Configure a jail: Any property can be reconfigured with the `set` subcommand.

To set the jail's `tag` property:

```
# iocage set tag=myjail 26e8e027
```

Get a jail property: To verify any property, use the `get` subcommand:

```
# iocage get tag 26e8e027
```

Get all properties, or display all supported properties:

```
# iocage get all 26e8e027
```

System-wide defaults: Starting with Version 1.6.0 of `iocage`, system-wide defaults can be set. These defaults will be reapplied for all newly created jails. To create a system-wide default override for a property, simply specify the default keyword instead of a jail `UUID` or `tag`.

To turn off VNET capability for all newly created jails:

```
# iocage set vnet=off default
```

Destroy a jail:

```
# iocage destroy myjail
```

The jail must be down or stopped before it can be destroyed.

Warning: This will irreversibly destroy the jail!

25.2.3 `iocage` Networking

Jails have multiple networking options based on what features are desired.

25.2.3.1 For IP Alias-Based Networking (Shared IP)

`iocage` will try to guess whether VNET support is available in the system, and if it is, will enable it by default for newly created jails.

Make sure VNET is disabled:

```
# iocage get vnet UUID | TAG
```

If set to "on", disable it:

```
# iocage set vnet=off UUID | TAG
```

A system-wide default can be configured if required. This will take effect for newly created jails only.

```
# iocage set vnet=off default
```

Configure an IP address:

```
# iocage set ip4_addr="em0|10.1.1.10/24" UUID| TAG
```

If multiple addresses are desired, just separate the configuration directives with a comma:

```
# iocage set ip4_addr="em0|10.1.1.10/24,em0|10.1.1.11/24" UUID|
  TAG
```

Start jail:

```
# iocage start UUID | TAG
```

Verify visible IP configuration in the jail (the jail must be running for this to work):

```
# iocage exec UUID | TAG ifconfig
```

25.2.3.2 For VNET Networking

Add bridge configuration to **/etc/rc.conf** (on the host):

```
cloned_interfaces="bridge0 bridge1"
ifconfig _ bridge0="addm em0 up"
ifconfig_em0="up"
```

Add these tunables to **/etc/sysctl.conf**:

```
net.inet.ip.forwarding=1
net.link.bridge.pfil _ onlyip=0
net.link.bridge.pfil _ bridge=0
net.link.bridge.pfil_member=0
```

Configure default gateway for jail:

```
# iocage set defaultrouter=10.1.1.254 UUID | TAG
```

Configure an IP address:

```
# iocage set ip4_addr="vnet0|10.1.1.10/24" UUID | TAG
```

Start jail and ping default gateway.

Start the jail:

```
# iocage start UUID | TAG
```

Open a console into the jail:

```
# iocage console UUID | TAG
```

Ping default gateway:

```
# ping 10.1.1.254
```

25.2.3.3 Configuring Network Interfaces

`iocage` handles network configuration for both shared IP and VNET jails transparently.

Configuring a shared IP jail:

```
IPV4
# iocage set ip4_addr="em0|192.168.0.10/24" UUID|TAG
IPV6
# iocage set ip6_addr="em0|2001:123:456:242::5/64" UUID|TAG
```

This command will add an IP alias of 192.168.0.10/24 to interface `em0` for the shared IP jail at start time, as well as 2001:123:456:242::5/64.

Configuring a VNET jail:

To configure both IPV4 and IPV6:

```
# iocage set ip4_addr="vnet0|192.168.0.10/24" UUID|TAG
# iocage set ip6_addr="vnet0|2001:123:456:242::5/64" UUID|TAG
# iocage set defaultrouter6="2001:123:456:242::1" UUID|TAG
```

For VNET jails, a default route has to be specified also.

Additionally, to start a jail with *no* IPV4/6 address, set these properties:

```
# iocage set ip4_addr=none ip6_addr=none UUID|TAG
# iocage set defaultrouter=none defaultrouter6=none UUID|TAG
```

25.2.4 Jail Types

`iocage` supports five different jail types:

- thick (default)
- thin
- base
- template
- empty

Full (thick): Full (thick) jail is the default type and it is created with the following command:

```
# iocage create
```

A full jail has a fully independent ZFS dataset suitable for network replication (ZFS send/recv).

Clone (thin): Thin jails are lightweight clones created with:

```
# iocage create -c
```

Thin jails are cloned from the appropriate release at creation time and consume only a fraction of the space, preserving only the changing data.

Base: `iocage` basejails use independent read-only ZFS file system clones to achieve the same functionality.

To create a basejail:

```
# iocage create -b
```

Basejails reclone their base file systems at each startup. They are ideal for environments where patching or upgrades are required at once to multiple jails.

Template: Template is simply a jail where the `template` property is set to "yes."

To turn a jail into a template:

```
# iocage set template=yes UUID|TAG
```

After this operation the jail can be listed with:

```
# iocage list -t
```

To deploy a jail from this template:

```
# iocage clone TEMPLATE_UUID tag=mynewjail
```

Templates can be converted back and forth by setting the `template` property.

Empty: Empty jails are intended for unsupported jail setups or testing.

To create an empty jail:

```
# iocage create -e
```

These are ideal for experimentation with unsupported releases or LINUX jails.

25.2.5 Best Practices

These are some generic guidelines for working with `iocage`-managed jails.

Using PF (PF packet filter firewall) as a module: This is the default setting in the generic kernel.

Always use an easily-recognizable `tag` for your jails and templates!

This will help you avoid mistakes and easily identify jails.

Set the notes property: Set the `notes` property to something meaningful, especially for templates and jails you might use only once in a while.

Check your firewall rules: When using IP or Internet Protocol firewall (IPFW) inside a VNET jail put `firewall _ enable="YES" firewall _ type="open"`

into **/etc/rc.conf** at the start. This way you can exclude the firewall from blocking. Lock it down once you have tested everything. Also check PF firewall rules on the host if you happen to mix both.

Get rid of old snapshots: Remove snapshots you do not need, especially from jails where data is changing a lot!

Use the `chroot` *subcommand*: In case you need to access or modify files in a template or a jail which is in a stopped state, use:

```
# iocage chroot UUID | TAG
```

This way, you do not need to start the jail or convert the template.

25.2.6 Advanced Usage

Clones: When a jail is cloned, `iocage` creates a ZFS clone file system. Clones are cheap lightweight snapshots.

A clone depends on its source snapshot and file system. If you wish to destroy the source jail and preserve its clones, you need to promote the clone first, otherwise the source jail cannot be destroyed.

Create a clone: Clone `myjail` to `myjail2` as follows:

```
# iocage clone myjail tag=myjail2
```

To clone a jail from an existing snapshot:

```
# iocage clone myjail@snapshotname tag=myjail3
```

Promoting a clone: To promote a cloned jail:

```
# iocage promote UUID | TAG
```

This step will reverse the clone and source jail relationship. Basically, the clone will become the source and the source jail will be demoted to a clone.

Then, you can remove the demoted jail with:

```
# iocage destroy UUID | TAG
```

Updating jails: Updates are handled with the `freebsd-update` utility. Jails can be updated while they are stopped or running.

To update a jail to the latest patch level:

```
# iocage update UUID | TAG
```

This will create a back-out snapshot of the jail automatically.

When you have finished updating and the jail is working correctly, simply remove the snapshot:

```
# iocage snapremove UUID|TAG@snapshotname
```

In the case where the update breaks the jail, simply revert back to the snapshot:

```
# iocage rollback UUID|TAG@snapshotname
```

If you wish to test updating without affecting a jail, create a clone and update it in the same way as outlined previously.

To clone a jail:

```
# iocage clone UUID|TAG tag=testupdate
```

Upgrading jails: Upgrades are handled with the `freebsd-update` utility. By default, the upgrade command will try to upgrade the jail to the host's release version (`uname -r`).

Based on the jail `type` property, upgrades are handled differently for basejails and nonbasejails.

Upgrading a nonbasejail: To upgrade a normal jail (nonbasejail) to the host's release:

```
# iocage upgrade UUID | TAG
```

This will upgrade the jail to the same release as the host.

To upgrade to a specific release:

```
# iocage upgrade UUID|TAG release=10.2-RELEASE
```

Upgrade basejail: To upgrade a basejail, first verify whether the jail is a basejail:

```
# iocage get type UUID|TAG
```

This command should return type "`basejail`".

The upgrade can be forced while the jail is online by the following:

```
# iocage upgrade UUID|TAG
```

This will forcibly reclone the basejail file systems while the jail is running (no downtime) and update the jails with the changes from the new release.

```
# iocage set release=10.1-RELEASE UUID|TAG
```

This will cause the jail to reclone its file systems from Release 10.1 on the next jail start. This will not update the jail's files with changes from the next release!

Autoboot:

Make sure `iocage _ enable="YES"` is set in **/etc/rc.conf**.

To enable a jail to autoboot during a boot:

```
# iocage set boot=on UUID|TAG
```

Boot priority: Boot order can be specified by setting the priority value:

```
# iocage set priority=20 UUID|TAG
```

A lower value means a higher boot priority.

Snapshot management: `iocage` supports transparent ZFS snapshot management out of the box. Snapshots are point-in-time copies of data; safety points to which a jail can be restored at any time. Initially, snapshots take up almost no space, as only changes in data is recorded.

To list snapshots for a jail:

```
# iocage snaplist UUID|TAG
```

To create a new snapshot run:

```
# iocage snapshot UUID|TAG
```

This will create a snapshot based on the current time.

To create a snapshot with a custom naming run:

```
# iocage snapshot UUID|TAG@mysnapshotname
```

Resource limits: `iocage` can enable optional resource limits for a jail. The outlined procedure should provide enough for a decent starting point.

Limit core or thread: Limit a jail to a single thread or core number 1:

```
# iocage set cpuset=1 UUID|TAG iocage start UUID|TAG
```

List applied rules: List applied limits:

```
# iocage limits UUID|TAG
```

Limit DRAM use: Limit a jail to 4G dynamic random-access memory (DRAM) use (limiting resident set size [RSS] memory use can be done on the fly):

```
# iocage set memoryuse=4G:deny UUID|TAG
```

Turn on resource limits: Turn on resource limiting for jail:

```
# iocage set rlimits=on UUID|TAG
```

Apply limits: Apply limit on the fly:

```
# iocage cap UUID | TAG
```

Check limits: Check active limits:

```
# iocage limits UUID | TAG
```

Limit central processing unit (CPU) use by %: Limit CPU execution to 20%:

```
# iocage set pcpu=20:deny UUID|TAG iocage cap UUID|TAG
```

Check limits:

```
# iocage limits UUID | TAG
```

Resetting a jail's properties: The `iocage reset` command resets jail properties. To reset to defaults:

```
# iocage reset UUID | TAG
```

You can also reset every jail to the default properties:

```
# iocage reset ALL
```

Resetting a jail will retain the jail's `UUID` and `TAG`. Everything else will be lost. Make sure to set any custom properties that you need.

Automatic package installation: Packages can be installed automatically at jail creation time. Specify the `pkglist` property at creation time, which should point to a text file containing one package name per line. Please note that you will need to have an Internet connection for this to work. `pkg install` will try to get the packages from online repositories.

Create a `pkgs.txt` file with your favorite text editor and add package names to it.

```
pkgs.txt:
nginx
```

Now simply create a jail and supply the **pkgs.txt** file:

```
# iocage create pkglist=/path-to/pkgs.txt tag=myjail
```

This will install `nginx` in the newly created jail.

25.2.7 How to Create and Use Templates

First, set up a custom jail, and then create a template from it. This is the expedient way to deploy all packages and preconfigured settings in the jail.

Any jail can be converted to a template and back to a jail again. A template is just another jail which has the `template` property set to "yes". The difference is that templates are not started by `iocage`.

To create a template with `iocage`:

1. Create a new jail.

   ```
   # iocage create tag=mytemplate
   ```

2. Configure the jail's networking to suit the way you want to deploy it.

3. Customize the jail.

4. When customization is completed, stop the jail.

   ```
   # iocage stop UUID | TAG
   ```

5. Good practices dictate that you set some notes to explain the jail.

   ```
   # iocage set notes="customized nginx jail" UUID |TAG
   ```

6. Turn the `template` property on.

   ```
   # iocage set template=yes UUID | TAG
   ```

7. List your template.

   ```
   # iocage list -t
   ```

To use the created template:

1. To create a new jail from the created template, clone it.

   ```
   # iocage clone UUID-of-mytemplate tag=mynewjail
   ```

2. List new jail.

   ```
   # iocage list
   ```

3. Start the new jail.

   ```
   # iocage start UUID | TAG
   ```

If you need to make further customization to the template or want to patch it, you have two options:

- Convert the template back to a jail with `iocage set template=no UUID-of-template`, and start the jail.
- If you do not need network access to make the changes, execute `iocage chroot UUID-of-template`, make the changes, and exit.

25.2.8 Create a Jail Package

A jail package is a small template which can be used on top of ordinary jails. The release and patch levels have to match between the package and an ordinary jail.

`iocage` uses the record function for this, which is a union file system (`unionfs`). The resulting package can be stored on a web server with a `checksum` file ready to be used anywhere.

To create a jail package, based on the Nginx web server, using `iocage`:

1. Create a new jail.

```
# iocage create -c tag=nginx
```

2. Start the jail.

```
# iocage start UUID | TAG
```

3. Configure networking to enable internet access for this jail.
4. Turn on recording.

```
# iocage record start UUID | TAG
```

From now on, every change will be recorded under **/iocage/jails/UUID/recorded**.

5. Install `nginx` with `pkg install nginx`.
6. Install any other software you might require. With just `nginx` installed, this is a very lightweight container!
7. Customize configuration files in the jail.
8. Once finished, stop recording changes with:

```
# iocage record stop UUID | TAG
```

Optionally, stop the jail.

9. Examine **/iocage/jails/UUID/recorded**:

```
# find /iocage/jails/UUID/recorded -type f
```

10. Remove any unnecessary files and make final customization/changes.
11. Execute the package create command.

```
# iocage package UUID | TAG
```

This will create a package in **/iocage/packages** with a `SHA256 checksum` file.

12. Optionally, discard the jail now.

```
# iocage destroy UUID | TAG
```

The resulting **UUID.tar.xz** can now be deployed, as follows.

1. Create a new jail.

```
# iocage create -c
```

2. Deploy package.

   ```
   # iocage import UUID tag=myjail
   ```

3. List the jail.

   ```
   # iocage list|grep myjail, grab UUID
   ```

4. Start the jail.

   ```
   # iocage start UUID | TAG
   ```

5. Examine your changes and packages.

25.2.9 iocage Installation and Worked Examples

The following are explicit installation instructions for iocage on PC-BSD, and three worked examples showing how to do some basic iocage operations to create, view, and manage jails.

Installation: Depending on your version and installation type of PC-BSD, you may need to install the iocage package before beginning the first step of installation as shown here. For example, on a newly built TrueOS PC-BSD system, the package was already installed and ready to be expanded. The easiest way for you to know that the package has already been installed on your PC-BSD system is to proceed with the first iocage fetch command shown here. If you get an error message, then execute the following command as superuser:

```
# pkg install iocage
```

Then, proceed on to the instructions for installation.

Note: Some of the output from the system is truncated to improve readability.

```
[bob@pcbsd-1233] /usr/home/bob# iocage fetch
Supported releases are:
   10.2-RELEASE
    9.3-RELEASE
Please select a release [10.2-RELEASE]: <Enter>
src.txz                                       100% of  118 MB  626
                                              kBps 03m14s
Extracting: base.txz
Extracting: doc.txz
Extracting: lib32.txz
Extracting: src.txz
* Updating base jail..
src component not installed, skipped
Looking up update.FreeBSD.org mirrors... none found.
Fetching public key from update.FreeBSD.org... done.
Fetching metadata signature for 10.2-RELEASE from update.FreeBSD.
org... done.
```

```
Fetching metadata index... done.
Fetching 2 metadata files... done.
Inspecting system…
…
/usr/share/man/man8/ntpq.8.gz
/usr/share/man/man8/sntp.8.gz
src component not installed, skipped
Installing updates... done.
[bob@pcbsd-1233] /usr/home/bob#
```

You have now installed `iocage` on your PC-BSD system.

EXERCISE 25.1

Follow the steps listed previously to install the latest release of `iocage` on your system.

Example 25.1: Creating Jails and Viewing Some of Their Properties

The following example allows you to create two jails and view some of their properties.
First jail:

```
[bob@pcbsd-1233] /usr/home/bob# iocage create
** interfaces=vnet0:bridge0,vnet1:bridge1
** vnet=off
** host_hostname=fabaf6b1-94c3-11e5-860b-08002738ba74
** hostname=fabaf6b1-94c3-11e5-860b-08002738ba74
** ip4_addr=none
...
[bob@pcbsd-1233] /usr/home/bob# iocage list
JID               UUID                    BOOT  STATE  TAG
-     527c21ff-9573-11e5-9cc9-08002738ba74  off   down
                                               2015-11-27@17:57:04
[bob@pcbsd-1233] /usr/home/bob#
```

The `TAG` is the name of the jail, and the `UUID` is another way of identifying it. For the next jail, we will create it with a more manageable `TAG` and also add an IPV4 address to it.

Second Jail: This command will name the new jail `bobsecond`, using the `tag` subcommand. It will also have an IP address of 192.168.0.100, which will be added as an alias by `iocage` when the jail is started, and removed when the jail is stopped. The alias will be added to the device network interface `em0`.
The `tag` allows you to refer to the newly created jail as `bobsecond`.

```
# iocage create tag=bobsecond ip4_addr="em0|192.168.0.100"
Configuring jail..
** interfaces=vnet0:bridge0,vnet1:bridge1
…
```

```
[bob@pcbsd-1233] /usr/home/bob# iocage list
JID                 UUID                     BOOT STATE TAG
-   44599667-9574-11e5-9cc9-08002738ba74 off  down  bobsecond
-   527c21ff-9573-11e5-9cc9-08002738ba74 off  down
                                                2015-11-27@17:57:04
```

The name of the new jail is bobsecond, with a reference UUID (44599667-9574-11e5-9cc9-08002738ba74).

EXERCISE 25.2

Similar to the steps shown in Example 25.1, create a lightweight jail on your system. Name or tag the jail with your own unique name. Then use the zfs list command to display the ZFS file system for this jail, and its mount point. How large is the jail in bytes?

Example 25.2: Listing Jails and Starting Them Automatically at System Boot

As shown in Example 25.1, the following command lists information about the current Jails:

```
# iocage list
JID                UUID                      BOOT STATE TA
-   44599667-9574-11e5-9cc9-08002738ba74 off  down  bobsecond
```

You can see that the boot flag is off. This means the jail will not be started at boot time. The STATE identifier means that the jail has not been started, or is inactive.

Starting jails automatically at system boot: To allow iocage to start your jails at boot time, add the following line to **/etc/rc.conf**:

```
iocage_enable="YES"
```

Individual jails can be marked for starting at system boot by using this command:

```
# iocage set boot=on bobsecond
```

You can specify the order for jails to start by setting their priority:

```
# iocage set priority=20 bobsecond
```

A lower value means a higher boot priority (i.e., boot the jail earlier).

EXERCISE 25.3

For the jail you created in in-chapter Exercise 25.2, change its properties so that it autostarts at system boot. Then reboot the system to see that the jail starts automatically.

Example 25.3: Looking Into, Starting, Stopping, and Destroying a Jail

You can `chroot` into your jail without starting or running the jail:

```
# iocage chroot bobsecond
root@pcbsd-1233:/ #
```

This can be useful for modifying the jail. Explore the jail with the `ls` command:

```
root@pcbsd-1233:/ # ls
.cshrc    bin      etc      media    rescue   sys      var
.profile  boot     lib      mnt      root     tmp
COPYRIGHT dev      libexec  proc     sbin     usr
root@pcbsd-1233:/ #
```

See how the network connection is configured:

```
root@pcbsd-1233:/ # ifconfig
em0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric
0 options=9b<RXCSUM,TXCSUM,VLAN_MTU,VLAN_HWTAGGING,VLAN_HWCSUM>
        ether 08:00:27:38:ba:74
        inet6 fe80::a00:27ff:fe38:ba74%em0 prefixlen 64
        scopeid 0x1
        inet 192.168.0.16 netmask 0xffffff00 broadcast
        192.168.0.255
        nd6 options=23<PERFORMNUD,ACCEPT_RTADV,AUTO_LINKLOCAL>
        media: Ethernet autoselect (1000baseT <full-duplex>)
        status: active
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> metric 0 mtu
16384
options=600003<RXCSUM,TXCSUM,RXCSUM_IPV6,TXCSUM_IPV6>
        inet6 ::1 prefixlen 128
        inet6 fe80::1%lo0 prefixlen 64 scopeid 0x2
        inet 127.0.0.1 netmask 0xff000000
        nd6 options=21<PERFORMNUD,AUTO_LINKLOCAL>
root@pcbsd-1233:/ #
```

To get out of the `chroot`, press <CTRL-D> or type **`exit`**.

Starting a jail

```
[bob@pcbsd-1233] /usr/home/bob# iocage start bobsecond
* Starting 44599667-9574-11e5-9cc9-08002738ba74 (bobsecond)
  + Started (shared IP mode) OK
  + Starting services        OK
[bob@pcbsd-1233] /usr/home/bob#
[bob@pcbsd-1233] /usr/home/bob# iocage list
JID          UUID                                 BOOT  STATE  TAG
```

```
1 44599667-9574-11e5-9cc9-08002738ba74  off   up     bobsecond
- 527c21ff-9573-11e5-9cc9-08002738ba74  off   down
                                                2015-11-27@17:57:04
[bob@pcbsd-1233] /usr/home/bob#
```

The jail named bobsecond is up.

Accessing a jail console: This command allows you access to the jail without logging into it:

```
[bob@pcbsd-1233] /usr/home/bob# iocage console bobsecond
FreeBSD 10.2-RELEASE-p4 (GENERIC) #0: Tue Aug 18 15:15:36 UTC
2015
Welcome to FreeBSD!
Release Notes, Errata: https://www.FreeBSD.org/releases/
...
root@44599667-9574-11e5-9cc9-08002738ba74:~ #
```

You are now in the jail as if you had logged in at the console. For example, we can now start sshd manually. Watch as it creates missing keys automatically:

```
root@44599667-9574-11e5-9cc9-08002738ba74:~ # service sshd
onestart
Generating RSA1 host key.
2048 09:ce:a8:b8:dc:5b:e6:22:9c:a8:f4:6b:76:da:c7:10
root@44599667-9574-11e5-9cc9-08002738ba74 (RSA1)
Generating RSA host key.
2048 a4:03:c5:61:76:ed:41:80:77:20:3f:d6:99:89:37:4d
root@44599667-9574-11e5-9cc9-08002738ba74 (RSA)
Generating DSA host key.
1024 78:4b:04:4b:e8:f5:09:a5:5f:0e:71:02:b4:c6:b2:7b
root@44599667-9574-11e5-9cc9-08002738ba74 (DSA)
Generating ECDSA host key.
256 c7:98:87:dc:16:37:df:95:21:d0:24:04:75:a8:50:c3
root@44599667-9574-11e5-9cc9-08002738ba74 (ECDSA)
Generating ED25519 host key.
256 53:08:ac:52:d3:3e:ba:53:9e:ec:0b:c6:42:22:bc:c8
root@44599667-9574-11e5-9cc9-08002738ba74 (ED25519)
Performing sanity check on sshd configuration.
Starting sshd.
root@44599667-9574-11e5-9cc9-08002738ba74:~ #
```

Check the network configuration:

```
root@44599667-9574-11e5-9cc9-08002738ba74:~ # ifconfig
em0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric
0 mtu 1500
```

```
        options=9b<RXCSUM,TXCSUM,VLAN_MTU,VLAN_HWTAGGING,
        VLAN_HWCSUM>
        ether 08:00:27:38:ba:74
        inet 192.168.0.100 netmask 0xffffffff broadcast
        192.168.0.100
        media: Ethernet autoselect (1000baseT <full-duplex>)
        status: active
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> metric 0 mtu
16384
        options=600003<RXCSUM,TXCSUM,RXCSUM_IPV6,TXCSUM_IPV6>
root@44599667-9574-11e5-9cc9-08002738ba74:~ #
```

Change the root password in the jail bobsecond while we are logged into its console:

```
root@44599667-9574-11e5-9cc9-08002738ba74:~ # passwd
Changing local password for root
New Password: xxxxx
Retype New Password: xxxxx
root@44599667-9574-11e5-9cc9-08002738ba74:~ #
```

To close the console connection into bobsecond, type **`exit`** or **`logout`**.
Restarting and stopping a jail:

```
[bob@pcbsd-1233] /usr/home/bob# iocage restart bobsecond
* Soft restarting 44599667-9574-11e5-9cc9-08002738ba74
(bobsecond)
[bob@pcbsd-1233] /usr/home/bob#
[bob@pcbsd-1233] /usr/home/bob# iocage stop bobsecond
* Stopping 44599667-9574-11e5-9cc9-08002738ba74 (bobsecond)
  + Running pre-stop         OK
  + Stopping services        OK
  + Removing jail process    OK
  + Running post-stop        OK
rctl: RACCT/RCTL present, but disabled; enable using kern.
racct.enable=1 tunable
[bob@pcbsd-1233] /usr/home/bob#
```

Destroying a jail:

If you wish to remove a jail from a system, you destroy it.
You also have to make sure the jail is stopped, or down!

First use `iocage list` to see what jails we have defined:

```
[bob@pcbsd-1233] /usr/home/bob# iocage list
```

```
JID             UUID                   BOOT STATE  TAG
2   44599667-9574-11e5-9cc9-08002738ba74 off  up     bobsecond
-   527c21ff-9573-11e5-9cc9-08002738ba74 off  down
                                                   2015-11-27@17:57:04
```

Then use `iocage destroy` to completely remove the first jail we created:

```
[bob@pcbsd-1233] /usr/home/bob# iocage destroy
2015-11-27@17:57:04
  WARNING: this will destroy jail 527c21ff-9573-11e5-9cc9-
08002738ba74
  Dataset: tank/iocage/jails/527c21ff-9573-11e5
-9cc9-08002738ba74
  Are you sure ? y[N]: y
  Destroying: 527c21ff-9573-11e5-9cc9-08002738ba74
[bob@pcbsd-1233] /usr/home/bob#
```

Notice that in the `iocage destroy` command, we used the `tag` of the first jail as its identifier.

EXERCISE 25.4

Create a jail, start it, and set properties on it so you can `ssh` into it from the host, and from other computers on your local area network (LAN).

25.3 SOLARIS ZONES VIRTUALIZATION METHOD

Solaris zones are used to virtualize OS services and provide an isolated and secure environment for running applications. A *nonglobal zone*, usually called just a zone, is a virtualized OS environment created within a single instance of the Solaris OS. The single instance of the OS is called the *global zone*. The global and any nonglobal zones share only one instance of the Solaris kernel.

25.3.1 Nonglobal Zone State Model

Before doing some example installations of zones, it is important to know what operating states a nonglobal zone can have. The following model details those states, as seen in Figure 25.1. Example commands that affect those states are also shown in the description of the model that follows:

A nonglobal zone can be in one of the following seven states, as shown in Figure 25.1:

Configured: The zone's configuration is complete and committed to stable storage. However, those elements of the zone's application environment that must be specified after initial boot are not yet present.

Incomplete: During an install or uninstall operation, `zoneadm` sets the state of the target zone to incomplete. On successful completion of the operation, the state is set to the correct state.

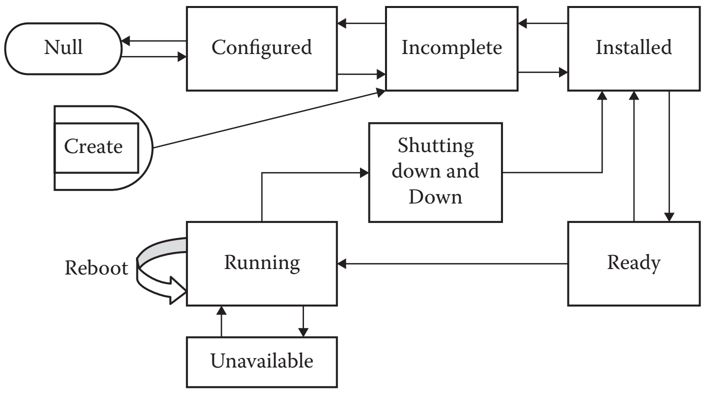


FIGURE 25.1 Nonglobal zone states.

A damaged installed zone can be marked incomplete by using the `mark` subcommand of `zoneadm`. Zones in the incomplete state are shown in the output of `zoneadm list -iv`.

Unavailable: Indicates that the zone has been installed, but cannot be verified, made ready, booted, attached, or moved. A zone enters the unavailable state at the following times:

When the zone's storage is unavailable and **svc:/system/zones:default** begins, such as during system boot

When the zone's storage is unavailable

When archive-based installations fail after successful archive extraction

When the zone's software is incompatible with the global zone's software, such as after an improper `-F` (force) attach

Installed: The zone's configuration is instantiated on the system. The `zoneadm` command is used to verify that the configuration can be successfully used on the designated Solaris system. Packages are installed under the zone's root path. In this state, the zone has no associated virtual platform.

Ready: The virtual platform for the zone is established. The kernel creates the *zsched process*, network interfaces are set up and made available to the zone, file systems are mounted, and devices are configured. A unique zone ID is assigned by the system. At this stage, no processes associated with the zone have been started.

Running: User processes associated with the zone application environment are running. The zone enters the running state as soon as the first user process associated with the application environment (`init`) is created.

Shutting down and Down: These states are transitional states that are visible while the zone is being halted. However, a zone that is unable to shut down for any reason will stop in one of these states.

25.3.2 Commands That Affect Zone State

The following table is a listing of commands and subcommands in Solaris that affect the zone states as described in the previous subsection. You can also see the `zonecfg` and `zoneadm` manual pages on the system for more description and explanation. The manual pages contain several examples. The general syntax of the `zoneadm` command is as follows:

SYNTAX

```
zoneadm -z zonename [-u uuid-match] subcommand [subcommand_options]
zoneadm [-R root] [-z zonename] [-u uuid-match] list [list_options]
zoneadm [-R root] -z zonename [-u uuid-match] mark incomplete
```

Purpose: The **`zoneadm`** utility is used to administer system zones. A zone is an application container that is maintained by the OS.

Commonly used options/features:

`zoneadm list -cv`	List zones with verbose output.
`zoneadm -z firstzone install`	Install a zone named **`firstzone`** with a ZFS.

Parameters changed through `zonecfg` do not affect a running zone. The zone must be rebooted for the changes to take effect.

25.3.3 Creating a Solaris Zone

Example 25.4: A First Zone

Objectives: To create a Solaris zone using the `zonecfg` and `zoneadm` commands, and to check its connection to the host with the `ping` command.

Introduction: A zone is a "container" which runs the same kernel as the host, but operates in an entirely autonomous and secure environment separate from the host. An illustration of the allowed states that a zone can be in at any time is given in Figure 25.1 and the commands that affect those states are shown in Table 25.1. Many of the steps in the example will take the zone through those states, and it would be instructive for you to refer back to this figure when you proceed through the steps of the example to verify what state the zone you are creating is currently in during any one step.

Prerequisites:

1. You must have superuser privileges on a Solaris system.
2. Your computer must be connected to the Internet, and you must have access to an Oracle IPS repository.

TABLE 25.1 zonecfg and zoneadm Subcommands and Arguments

Command, Arguments, and Subcommands	Brief Description of What it Does
Configured	
`zonecfg -z zonename verify`	
`zonecfg –z zonename set old=new`	Renames old `zonename` to new.
`zonecfg -z zonename commit`	
`zonecfg -z zonename delete`	
`zoneadm -z zonename attach`	
`zoneadm -z zonename verify`	
`zoneadm -z zonename install`	
`zoneadm -z zonename clone`	
`zoneadm -z zonename mark incomplete`	
`zoneadm -z zonename mark unavailable`	
Incomplete	
`zoneadm -z zonename uninstall`	
Unavailable	
`zoneadm -z zonename uninstall`	Uninstalls the zone from the specified system.
`zoneadm -z zonename attach`	
`zonecfg -z zonename`	Can be used to change `zonepath` and any other property or resource that can be changed when in the installed state.
Installed	
`zoneadm -z zonename ready` (optional)	
`zoneadm -z zonename boot`	
`zoneadm -z zonename uninstall`	Uninstalls the configuration of the specified zone from the system.
`zoneadm -z zonename move path`	
`zoneadm -z zonename detach`	
`zonecfg -z zonename`	Adds or removes an `attr`, `bootargs`, `capped-cpu`, `capped-memory`, `dataset`, `dedicated-cpu`, `device`, `fs`, `ip-type`, `limitpriv`, `net`, `rctl`, or `scheduling-class` property.
`zoneadm -z zonename mark incomplete`	
`zoneadm -z zonename mark unavailable`	
Ready	
`zoneadm -z zonename boot`	
`zoneadm halt`	Returns a zone in the ready state to the installed state.
`zonecfg -z zonename`	Adds or removes `attr`, `bootargs`, `capped-cpu`, `capped-memory`, `dataset`, `dedicated-cpu`, `device`, `fs`, `ip-type`, `limitpriv`, `net`, `rctl`, or `scheduling-class` property.
Running	
`zlogin options zonename`	
`zoneadm -z zonename reboot`	
`zoneadm -z zonename halt`	Returns a ready zone to the installed state.

TABLE 25.1 (CONTINUED) `zonecfg` and `zoneadm` Subcommands and Arguments

Command, Arguments, and Subcommands	Brief Description of What it Does
`zoneadm halt`	Returns a zone in the running state to the installed state.
`zoneadm -z shutdown`	Cleanly shuts down the zone.
`zonecfg -z zonename`	Adds or removes an `anet`, `attr`, `bootargs`, `capped-cpu`, `capped-memory`, `dataset`, `dedicated-cpu`, `device`, `fs`, `ip-type`, `limitpriv`, `net`, `rctl`, or `scheduling-class` property. The `zonepath` resource cannot be changed.

3. You must know the IP address of your host. You can find this out by using the `ipadm` command as follows:

```
root@solaris:~# ipadm show-addr
ADDROBJ         TYPE   STATE   ADDR
lo0/v4        static    ok     127.0.0.1/8
net0/v4         dhcp    ok     192.168.0.13/24
lo0/v6        static    ok     ::1/128
net0/v6     addrconf    ok     fe80::e611:5bff:fe12:c277/10
```

In the output on our system, the network interface card (NIC) is `net0`, and the IP address of the host is 192.168.0.13.

Procedures: Do the following steps, in the order presented here, to meet the objectives of this example.

1. Check what zones currently exist by listing their ZFS data sets.

```
root@solaris:~# zfs list | grep zones
rpool/VARSHARE/zones     144K   890G   144K  /system/zones
```

2. Check if virtual NICs exist.

```
root@solaris:~# dladm show-link
LINK             CLASS     MTU    STATE    OVER
net0              phys     1500    up        --
vboxnet0          phys     1500    up        --
```

3. Begin creation of the first zone using the `zonecfg` command.

```
root@solaris:~# zonecfg -z firstzone
Use 'create' to begin configuring a new zone.
```

4. You are then in the `zonecfg` program. Set important default options as shown, and then exit.

```
zonecfg:firstzone> create
create: Using system default template 'SYSdefault'
zonecfg:firstzone> set zonepath=/zones/firstzone
zonecfg:firstzone> set autoboot=true
```

```
zonecfg:firstzone> set bootargs="-m verbose"
zonecfg:firstzone> verify
zonecfg:firstzone> commit
zonecfg:firstzone> exit
```

5. Check the status of the virtual NIC.

```
root@solaris:~# dladm show-link
LINK            CLASS     MTU    STATE    OVER
net0             phys    1500      up       --
vboxnet0        phys     1500      up       --
```

6. Check the configured status of the zone.

```
root@solaris:~# zoneadm list -cv
ID NAME      STATUS     PATH               BRAND    IP
0 global     running     /                 solaris  shared
- firstzone configured   /zones/firstzone  solaris  excl
```

7. Install the zone, which we will name `firstzone`, from an online IPS repository using the `zoneadm` command. Your computer must be connected to the Internet, and you must have access to an IPS repository. This is a long step (on our system it took about 20 minutes) where you have to wait for the repository packages to load and install.

```
root@solaris:~# zoneadm -z firstzone install
The following ZFS file system(s) have been created:
    rpool/zones
    rpool/zones/firstzone
Progress being logged to /var/log/zones/
zoneadm.20141017T083049Z.firstzone.install
       Image: Preparing at /zones/firstzone/root.
 Install Log: /system/volatile/install.3819/install_log
 AI Manifest: /tmp/manifest.xml.MVaWBh
  SC Profile: /usr/share/auto_install/sc_profiles/enable_
sci.xml
    Zonename: firstzone
Installation: Starting ...
        Creating IPS image
Startup linked: 1/1 done
        Installing packages from:
            solaris
                origin:  http://pkg.oracle.com/solaris/
release/
DOWNLOAD      PKGS     FILES        XFER (MB)     SPEED
Completed     282/282  53274/53274  351.9/351.9  344k/s
PHASE                            ITEMS
Installing new actions           71043/71043
Updating package state database  Done
Updating package cache           0/0
Updating image state              Done
Creating fast lookup database     Done
Updating package cache            1/1
Installation: Succeeded
        Note: Man pages can be obtained by installing
       pkg:/system/manual
done.
        Done: Installation completed in 1297.609 seconds.
  Next Steps: Boot the zone, then log into the zone
console (zlogin -C)
              to complete the configuration process.
Log saved in non-global zone as /zones/firstzone/root/var/
log/zones/zoneadm.20141017T083049Z.firstzone.install
```

8. Check the status of the zone.

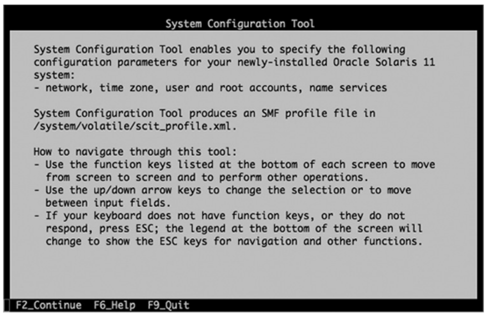
```
root@solaris:~# zoneadm list -iv
ID NAME      STATUS    PATH                BRAND    IP
0 global     running   /                   solaris  shared
- firstzone installed /zones/firstzone    solaris  excl
```

9. If you again check the zone datasets with `zfs list`, you can see how they have been augmented to reflect the addition of the zone.

```
root@solaris:~# zfs list | grep zones
rpool/VARSHARE/zones  144K   889G   144K  /system/zones
rpool/zones           858M   889G   152K  /zones
rpool/zones/firstzone 857M   889G   152K  /zones/firstzone
rpool/zones/firstzone
/rpool                857M   889G   144K  /rpool
rpool/zones/firstzone/
rpool/ROOT            857M   889G   144K  legacy
rpool/zones/firstzone/
rpool/ROOT/solaris    857M   889G   789M  /zones/
firstzone/root
rpool/zones/firstzone/
rpool/ROOT/solaris/var 67.8M 889G  66.8M /zones/
                                               firstzone/
root/var
rpool/zones/firstzone
/rpool/VARSHARE       144K   889G   144K  /var/share
rpool/zones/firstzone/
rpool/export          288K   889G   144K  /export
rpool/zones/firstzone/
rpool/export/home     144K   889G   144K  /export/home
```

10. Boot, or start, the zone and complete its system configuration using the system configuration tool. The first window which appears is shown in Figure 25.2.

    Note that, at the end of the creation process, you should type **~.** so that you do not log into the console.



```
System Configuration Tool

System Configuration Tool enables you to specify the following
configuration parameters for your newly-installed Oracle Solaris 11
system:
- network, time zone, user and root accounts, name services

System Configuration Tool produces an SMF profile file in
/system/volatile/scit_profile.xml.

How to navigate through this tool:
- Use the function keys listed at the bottom of each screen to move
  from screen to screen and to perform other operations.
- Use the up/down arrow keys to change the selection or to move
  between input fields.
- If your keyboard does not have function keys, or they do not
  respond, press ESC; the legend at the bottom of the screen will
  change to show the ESC keys for navigation and other functions.

F2_Continue  F6_Help  F9_Quit
```

FIGURE 25.2 System configuration tool window.

```
root@solaris:~# zoneadm -z firstzone boot; zlogin -C
firstzone
[Connected to zone 'firstzone' console]
Loading smf(5) service descriptions: 134/134
```

When the system configuration tool first window opens on screen, press <F2> to continue. Then, in the subsequent windows, enter the following:

```
Computer Name: firstzone     Press <F2>

IP Address: 192.168.0.25                        Press <F2>
Do Not Configure DNS                            Press <F2>
Alternate Name Service: Make the None choice.Press <F2>
Region: Where you are in the world.             Press <F2>
Time Zones: Your time zone                      Press <F2>
Root Password, User Name and Password: Your choice
                                                Press <F2>
Verify that the configuration is the way you want it.
                                                Press <F2>
```

The Configuration Tool window closes.

```
Booting continues…
<Output truncated...>
firstzone console login: ~.
[Connection to zone 'firstzone' console closed]
```

11. Check that the zone is booted and running.

```
root@solaris:~# zoneadm list -v
ID NAME       TATUS    PATH                BRAND     IP
0 global     running    /                  solaris   shared
1 firstzone  running   /zones/firstzone    solaris   excl
```

12. Check that the virtualized network interface card (VNIC) has been created.

```
root@solaris:~# dladm show-link
LINK                CLASS    MTU    STATE    OVER
net0                 phys    1500     up       --
vboxnet0             phys    1500     up       --
```

```
firstzone/net0     vnic    1500      up     net0
```

13. Before logging into the new zone, check that it is running and the NIC name.

```
root@solaris:~# zoneadm list -v
ID NAME      STATUS    PATH                 BRAND     IP
0 global     running    /                  solaris    shared
1 firstzone  running    /zones/firstzone solaris    excl
root@solaris:~# dladm show-link
LINK               CLASS     MTU    STATE    OVER
net0                phys     1500     up        --
vboxnet0            phys     1500     up        --
firstzone/net0      vnic     1500     up      net0
```

14. Log into the new zone, check the name of the OS.

```
root@solaris:~# zlogin firstzone
[Connected to zone 'firstzone' pts/2]
Oracle Corporation      SunOS 5.11  11.2  June 2014
root@firstzone:~# uname -a
SunOS firstzone 5.11 11.2 i86pc i386 i86pc
```

15. Check what IP address is assigned to `firstzone`. This would be the IP address you use to access this zone from the host machine, your LAN, or the Internet. The first three fields of the IP address, 192.168.0, indicate the subnetwork.

```
root@firstzone:~# ipadm show-addr
ADDROBJ       TYPE      STATE    ADDR
lo0/v4        static      ok     127.0.0.1/8
net0/v4       dhcp        ok     192.168.0.25/24
lo0/v6        static      ok      ::1/128
net0/v6       addrconf    ok    fe80::8:20ff:fe86:6094/10
```

16. This command shows the automatically created `net0` VNIC.

```
root@firstzone:~# dladm show-link
LINK                 CLASS     MTU    STATE    OVER
net0                 vnic      1500     up        ?
```

17. List the ZFS datasets available from within the zone itself.

```
root@firstzone:~# zfs list
NAME              USED      AVAIL     REFER  MOUNTPOINT
rpool             899M      889G      144K     /rpool
rpool/ROOT        897M      889G      144K     legacy
rpool/ROOT/
solaris           897M      889G      819M     /
rpool/ROOT/solaris/
var               68.7M     889G      67.5M   /var
rpool/VARSHARE   1.41M     889G      1.12M   /var/share
rpool/VARSHARE/
pkg               296K      889G      152K    /var/share/pkg
```

```
rpool/VARSHARE/pkg/
repositories     144K     889G     144K     /var/share/pkg/
repositories
rpool/export     456K     889G     152K     /export
rpool/export/
home             304K     889G     152K     /export/home
rpool/export/
home/bob         152K     889G    152K      /export/home/bob
```

18. Log out of zone `firstzone`.

```
root@firstzone:~# exit
logout
[Connection to zone 'firstzone' pts/2 closed]
```

19. Log into `firstzone` again.

```
root@solaris:~# zlogin firstzone
[Connected to zone 'firstzone' pts/2]
Oracle Corporation      SunOS 5.11   11.2   June 2014
root@firstzone:~#
```

20. Ping the host.

```
root@firstzone:~# ping 192.168.0.25
192.168.0.25 is alive
```

21. Log out of firstzone.

```
root@firstzone:~# exit
logout
[Connection to zone 'firstzone' pts/2 closed]
root@solaris:~#
```

Conclusions: With the `zonecfg` and `zoneadm` commands, we created a new zone named `firstzone`, and were able to check its connection to the host system.

EXERCISE 25.5

Make a drawing of Figure 25.1 on a piece of paper, at a somewhat larger scale than shown in the book. Then place the step number from Example 25.4 that put the zone in any of the states shown in Figure 25.1 on your drawing.

25.3.4 Installing a Web Server Application in a Zone

Example 25.5: Installing a Web Server Application in a Zone

Objectives: To create a new Solaris zone intended for installation of a web server application, and to install the Apache web server application into the zone.

Introduction: Using the same methods as in Example 25.4, we create a new Solaris zone, named `appzone`. Then, we show how to install the Apache web server into

that new zone. The example details the use of the `pkg install` command to do the installation.
We do not show the detailed use of the Apache web server.
Prerequisites:

1. You must have completed Example 25.4.
2. Your Solaris system must be connected to the Internet, and you must have access to an Oracle Solaris IPS online repository.

Procedures: Do the following steps, in the order presented here, to meet the objectives of this example.

1. Create a zone, named `appzone`, using a minimum amount of information.

```
root@solaris:~# zonecfg -z appzone "create ; set
                zonepath=/zones/appzone"
```

2. Use the `zonecfg` command to get information about the new zone.

```
root@solaris:~# zonecfg -z appzone info
zonename: appzone
zonepath: /zones/appzone
…
Output truncated...
```

3. Install the zone, as in Example 25.4, from an online Oracle IPS repository.

```
root@solaris:~# zoneadm -z appzone install
The following ZFS file system(s) have been created:
    rpool/zones/appzone
Progress being logged to /var/log/zones/
zoneadm.20141017T110817Z.appzone.install
       Image: Preparing at /zones/appzone/root.
 Install Log: /system/volatile/install.4635/install_log
 AI Manifest: /tmp/manifest.xml.YJa4bj
  SC Profile: /usr/share/auto_install/sc_profiles/enable_
sci.xml
    Zonename: appzone
Installation: Starting ...
        Creating IPS image
Startup linked: 1/1 done
        Installing packages from:
            solaris
                origin:  http://pkg.oracle.com/solaris/
release/
DOWNLOAD          PKGS         FILES    XFER (MB)   SPEED
Completed      282/282   53274/53274  351.9/351.9  3.4M/s
```

```
PHASE                                              ITEMS
Installing new actions                       71043/71043
Updating package state database                     Done
Updating package cache                               0/0
Updating image state                                Done
Creating fast lookup database                       Done
Updating package cache                               1/1
Installation: Succeeded
        Note: Man pages can be obtained by installing
pkg:/system/manual
 done.
        Done: Installation completed in 346.285 seconds.
  Next Steps: Boot the zone, then log into the zone
console (zlogin -C)
              to complete the configuration process.
Log saved in non-global zone as /zones/appzone/root/var/
log/zones/zoneadm.20141017T110817Z.appzone.install
```

4. Boot, or start, the new zone, log into it, and configure it with the same settings as Example 25.3. But, this time, use a different IP address: 192.168.0.26. Recall from Example 25.3 that the first three fields of the IP address of the new zone should be the same as the first three fields of the host IP address. This will put the zone on the same subnetwork as the host, which is mandatory if you want to access the application in the zone through the host NIC. Remember to leave the login console by typing **~.** on the command line.

```
root@solaris:~# zoneadm -z appzone boot; zlogin -C appzone
[Connected to zone 'appzone' console]
Loading smf(5) service descriptions: 134/134
System Configuration window opens...
Make your choices...
SC profile successfully generated as:
/etc/svc/profile/sysconfig/sysconfig-20141017-112005/
sc_profile.xml
Exiting System Configuration Tool. Log is available at:
/system/volatile/sysconfig/sysconfig.log.6274
<Output truncated...>
Oct 17 04:25:09 appzone sendmail[8419]: unable to qualify
my own domain name (appzone) -- using short name
appzone console login: ~.
[Connection to zone 'appzone' console closed]
```

5. Log into the zone with `zlogin`.

```
root@solaris:~# zlogin appzone
[Connected to zone 'appzone' pts/2]
Oracle Corporation      SunOS 5.11  11.2  June 2014
```

6. Obtain package information about the Apache web server software from the repository available.

```
root@appzone:~# pkg info -r /web/server/apache-22
          Name: web/server/apache-22
       Summary: Apache Web Server V2.2
   Description: The Apache HTTP Server Version 2.2
      Category: Web Services/Application and Web Servers
         State: Not installed
     Publisher: solaris
       Version: 2.2.27
 Build Release: 5.11
        Branch: 0.175.2.0.0.42.1
Packaging Date: June 23, 2014 02:28:11 AM
          Size: 9.19 MB
          FMRI: pkg://solaris/web/server/apache-
22@2.2.27,5.11-0.175.2.0.0.42.1:20140623T022811Z
```

7. Install the application package while in this zone.

```
root@appzone:~# pkg install /web/server/apache-22
           Packages to install:  8
            Services to change:  2
       Create boot environment: No
Create backup boot environment: No
DOWNLOAD   PKGS    FILES    XFER (MB)   SPEED
Completed   8/8   680/680    9.5/9.5    387k/s
PHASE                                              ITEMS
Installing new actions                             945/945
Updating package state database                    Done
Updating package cache                             0/0
Updating image state                               Done
Creating fast lookup database                      Done
Updating package cache                             1/1
```

8. Check on the installed application package.

```
root@appzone:~# pkg info /web/server/apache-22
          Name: web/server/apache-22
       Summary: Apache Web Server V2.2
   Description: The Apache HTTP Server Version 2.2
      Category: Web Services/Application and Web Servers
         State: Installed
     Publisher: solaris
       Version: 2.2.27
 Build Release: 5.11
        Branch: 0.175.2.0.0.42.1
Packaging Date: June 23, 2014 02:28:11 AM
          Size: 9.19 MB
          FMRI: pkg://solaris/web/server/apache-
22@2.2.27,5.11-0.175.2.0.0.42.1:20140623T022811Z
```

9. Check the IP address of the new zone.

```
root@appzone:~# ipadm show-addr
ADDROBJ       TYPE     STATE        ADDR
lo0/v4        static     ok        127.0.0.1/8
net0/v4       dhcp       ok        192.168.0.26/24
lo0/v6        static     ok        ::1/128
net0/v6       addrconf   ok
fe80::8:20ff:fed3:3836/10
```

10. Leave the new zone and return to the global zone.

```
root@appzone:~# exit
logout
[Connection to zone 'appzone' pts/2 closed]
root@solaris:~#
```

Conclusions: We created a new Solaris zone and downloaded and installed the Apache web server application into it.

EXERCISE 25.6

Install the package `gnu-emacs` into the zone named `appzone` created in Example 25.5.

25.4 VIRTUALBOX

VirtualBox is a software virtualization tool that can be used to install other OSs as *guests* on a *host* computer system, running either of our two base UNIX systems. We first discuss what advantages the VirtualBox method of virtualization provides, and then define some basic terms used in it. Finally, using an example-based format, we detail the installation and basic usage of VirtualBox on PC-BSD and Solaris.

Why use VirtualBox virtualization?

The advantages of the kind of virtual environment that VirtualBox gives you are:

- You can run multiple OSs *simultaneously*. VirtualBox allows you to run more than one OS at a time. You can switch between OS environments by simply moving your mouse into the window that contains one of the operating systems!
- From the system security point of view, you can isolate a particular application, such as a Web server or FTP site inside a VirtualBox guest, and anything that intrudes on its operation from the Internet only infects the guest system. VirtualBox is similar to using the PC-BSD iocage and jails, and Solaris zones.
- Once installed, a VM and its virtual hard disks can be considered a "sandbox" that can be arbitrarily frozen, unfrozen, copied, backed up, and moved between hosts. With "snapshots," you can save a particular state of a VM and revert back to that state, if necessary.

When dealing with VirtualBox, the following basic terminology is important:

Host OS: This is the operating system of the physical computer on which VirtualBox was installed. In our next examples, the hosts can be either PC-BSD, Solaris, Ubuntu LINUX, or Windows.

We show VirtualBox on LINUX and Windows host computers because there are many advantages to using these systems as hosts, and deploying our PC-BSD and Solaris UNIX systems on them as guests. We speak about some of these advantages later.

Guest OS: This is the operating system that is running inside the VM. VirtualBox can run any Intel or AMD x86 OS, but only officially supports and is optimized for a limited number of OSs, Solaris being one of them. The FreeBSD project, together with PC-BSD, has ported VirtualBox to the latest version of both OSs, and it can be successfully deployed on either of those systems as well. As we show here, it is possible to use LINUX and Windows computers as hosts to deploy PC-BSD and Solaris guest OSs on them.

VM: This is the special environment that VirtualBox creates for your guest OS while it is running. You run your guest OS via the use of a VM. A VirtualBox VM appears as a window on your computer's desktop in both PC-BSD and Solaris.

If you install VirtualBox as a host on a LINUX, Windows, or Apple OS X computer, you can install PC-BSD and/or Solaris as a VM on that host computer very easily. In fact, we show examples of just such a situation.

Two ways of controlling the settings of the VM are via the VirtualBox manager window, and via the VBoxManage command line program. A VM is essentially what you can see in its settings dialog via these two controlling facilities.

Guest additions: This refers to special software packages which are shipped with VirtualBox but designed to be installed inside a VM to improve performance of the guest OS and to add extra features. A good example of this is the setting for a bidirectional system paste buffer, which allows you to copy text in a guest window, and paste it into a host window, or vice versa. We show how to do this in example form.

We found that in both PC-BSD and Solaris, we needed to completely create a new VM and start it before we could enable the guest additions for that VM. This is done after the VM has started by making the VirtualBox menu choice "Devices > Insert Guest Additions CD". VirtualBox will then place/mount a "virtual" CD on the desktop of the VM, and if you click on it to launch the CD, you can install the contents of the virtual CD.

25.4.1 Installing and Running VirtualBox on a PC-BSD Host OS

As of the latest PC-BSD available at the time of the writing of this book, VirtualBox does *not* come preinstalled on PC-BSD.

The easiest way to install VirtualBox is to use the App Café to download and automatically install it on your system. In App Café, use the search icon at the top of the App Café window and enter VirtualBox as the search criterion.

You will then be presented with a choice of **`pcbsd-meta-virtualbox- 1429971087`**, or the latest available metapackage for VirtualBox. Click the "Install" button in App Café to install this package on your PC-BSD machine. After installation is complete, you will then have to log out and log back in to the computer so that you are added to the VirtualBox group.

To run VirtualBox, you can either click on its icon on the PC-BSD KDE desktop, or type **`VirtualBox`** on the command line in a console window.

25.4.2 Installing and Running Solaris VirtualBox

The following example shows the steps necessary to install VirtualBox on a Solaris system as a host system, in preparation for installing guest systems in it.

Example 25.6

Objectives: To install and run VirtualBox VM on a host Solaris system.

Introduction: We obtain the latest SunOS version of VirtualBox from www.virtualbox.org, and then install it. This can be done for other supported OSs in a very similar manner to allow you to have VirtualBox installed on those OSs. Check at www.virtualbox.org for supported OSs, and the methods of installing VirtualBox on them.

Prerequisites:

1. You must be working on a Solaris system.
2. You must have superuser privilege on the system.

Procedures: Do the following steps, in the order presented here, to meet the objectives of this example.

1. The latest version of VirtualBox can be found at www.virtualbox.org. In a web browser, download it. For example, at the time of the writing of this book, we downloaded **VirtualBox-4.3.16-95972-SunOS.tar.gz** via our Firefox Web browser. In our case, it was downloaded into the **Downloads** subdirectory of **/home/bob**.
2. Find out the name(s) of the current zones available on your system. On ours, only the global zone was available at the outset.

```
$ zonename
global
```

3. Become the superuser.

```
$ su
Password: xxx
```

4. Change the current working directory to the directory to which you downloaded VirtualBox, and list the names of the files there. You should find **VirtualBox-4.3.16-95972-SunOS.tar.gz** there.

```
# cd Downloads
~/Downloads# ls
IMG_20140604_105332_577.jpg  IMG_20140607_105110_564.jpg
IMG_20140607_104953_765.jpg  VirtualBox-4.3.16-95972-
                             SunOS.tar.gz
```

5. Unzip the VirtualBox tarred/zipped file.

```
~/Downloads# gunzip -cd VirtualBox-4.3.16-95972-SunOS.tar.
             gz | tar xvf -
x VirtualBox-4.3.16-SunOS-amd64-r95972.pkg, 214329344
bytes, 418612 tape blocks
x LICENSE, 20137 bytes, 40 tape blocks
x autoresponse, 151 bytes, 1 tape blocks
x ReadMe.txt, 1525 bytes, 3 tape blocks
```

6. List the file names after unzipping is completed.

```
~/Downloads# ls
autoresponse
ReadMe.txt
LICENSE
VirtualBox-4.3.16-95972-SunOS.tar.gz
VirtualBox-4.3.16-SunOS-amd64-r95972.pkg
```

7. Use the pkgadd command to add the VirtualBox software to the system. Be sure to answer `all`, and `y` for "yes" to the prompts that appear as shown.

```
~/Downloads# pkgadd -d VirtualBox-4.3.16-SunOS-
             amd64-r95972.pkg
The following packages are available:
  1  SUNWvbox     Oracle VM VirtualBox
                  (i386) 4.3.16,REV=2014.09.09.18.16.95972
Select package(s) you wish to process (or 'all' to process
all packages). (default: all) [?,??,q]: all
```

Processing package instance <SUNWvbox> from </home/bob/Downloads/VirtualBox-4.3.16-SunOS-amd64-r95972.pkg>

```
Oracle VM VirtualBox(i386) 4.3.16,REV=2014.09.09.18.16.959
72
Oracle Corporation
## Executing checkinstall script.
Checking package dependencies...
Done.
## Processing package information.
## Processing system information.
## Verifying disk space requirements.
## Checking for conflicts with packages already installed.
## Checking for setuid/setgid programs.
```

The following files are being installed with setuid and/or setgid permissions:

```
  /opt/VirtualBox/amd64/VBoxHeadless <setuid root>
  /opt/VirtualBox/amd64/VBoxNetAdpCtl <setuid root>
  /opt/VirtualBox/amd64/VBoxNetDHCP <setuid root>
  /opt/VirtualBox/amd64/VBoxNetNAT <setuid root>
  /opt/VirtualBox/amd64/VBoxSDL <setuid root>
  /opt/VirtualBox/amd64/VirtualBox <setuid root>
Do you want to install these as setuid/setgid files
[y,n,?,q] y
```

This package contains scripts which will be executed with superuser permission during the process of installing this package.

```
Do you want to continue with the installation of
<SUNWvbox> [y,n,?] y
Installing Oracle VM VirtualBox as <SUNWvbox>
## Installing part 1 of 1.
/etc/hostname.vboxnet0
/opt/VirtualBox/64/VBoxPython.so
/opt/VirtualBox/64/VBoxPython2_4.so
/opt/VirtualBox/64/VBoxPython2_6.so
/opt/VirtualBox/LICENSE
/opt/VirtualBox/UserManual.pdf
Output truncated...
[ verifying class <manifest> ]
## Executing postinstall script.
Checking for older bits...
Installing new ones...
Detected Solaris 11 Version 175
Loading VirtualBox kernel modules...
   - Loaded: Host module
   - Loaded: NetAdapter module
   - Loaded: NetFilter (Crossbow) module
   - Loaded: USBMonitor module
   - Loaded: USB module
Configuring services...
   - Loaded: Zone access service
Installing MIME types and icons...
Installing Python bindings...
   - Installed: Bindings for Python 2.6
Updating the boot archive...
Installation of <SUNWvbox> was successful.
```

8. Exit from superuser.

```
~/Downloads# exit
$
```

9. To run VirtualBox, on the Solaris desktop make the pull-down menu choice "Applications > System Tools > Oracle VM VirtualBox". Alternatively, you can type **`VirtualBox`** as superuser on the command line in a console window to launch the program.

Conclusion: We downloaded and installed the latest version of VirtualBox from www.virtualbox.org.

25.4.3 Installing a VM Guest

With VirtualBox installed on your host system, there is a very patented and universal way of creating VMs and then installing other OSs in those VMs. The following example shows the steps necessary to install a Solaris system in a guest VM running on a PC-BSD system. The technique shown is easily extended to allow you to install any supported OS as a guest VM in any supported host environment. We show examples of this scenario in Section 25.4.5.

Example 25.7: Creating a Solaris Virtual Machine Guest on a PC-BSD Host

Objectives: Using VirtualBox that has been installed on a PC-BSD host to create a VM guest running the Solaris OS.

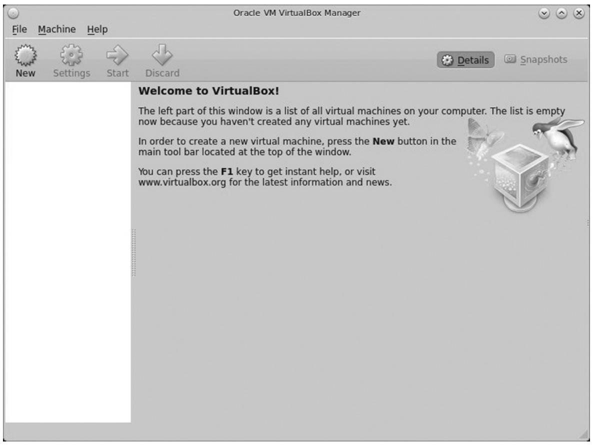
Introduction: The advantages of running two different OS kernels simultaneously on the same hardware are evident when you can utilize the stronger features of one or more of them in virtual environments to provide secure services over a LAN or the Internet. One of the lessons to be learned from this example is that resources such as memory and disk space are shared between host and guest(s), thereby affecting the performance of each.

Prerequisites:

1. You must have VirtualBox installed on your PC-BSD system.
2. You must have installation media in the form of a DVD or an International Organization for Standardization (ISO) file for Solaris, or any other supported OS you want to substitute for it in the VM guest we create in this example.

Procedures: Do the following steps, in the order presented here, to meet the objectives of this example.

1. Once installed on PC-BSD, start VirtualBox by typing **`VirtualBox`** as superuser on the command line in a console window. When VirtualBox launches for the first time, the screen shown in Figure 25.3 appears.



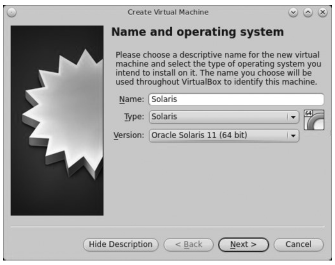
Welcome to VirtualBox!

The left part of this window is a list of all virtual machines on your computer. The list is empty now because you haven't created any virtual machines yet.

In order to create a new virtual machine, press the **New** button in the main tool bar located at the top of the window.

You can press the **F1** key to get instant help, or visit www.virtualbox.org for the latest information and news.

FIGURE 25.3 VirtualBox VM manager screen.

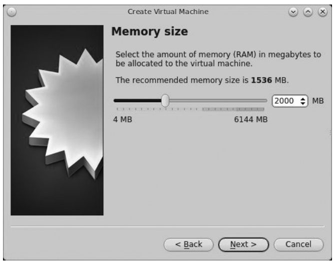


Name and operating system

Please choose a descriptive name for the new virtual machine and select the type of operating system you intend to install on it. The name you choose will be used throughout VirtualBox to identify this machine.

Name: Solaris
Type: Solaris
Version: Oracle Solaris 11 (64 bit)

FIGURE 25.4 Creating VM wizard.



Memory size

Select the amount of memory (RAM) in megabytes to be allocated to the virtual machine.

The recommended memory size is **1536** MB.

2000 MB

4 MB 6144 MB

FIGURE 25.5 Selecting the amount of memory reserved for the VM.

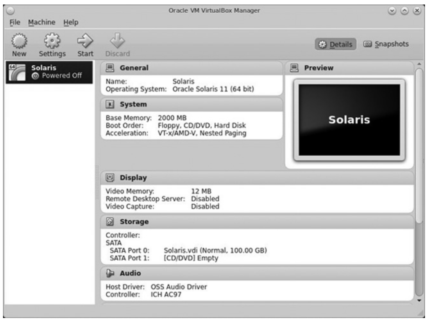
In any of the VirtualBox manager or wizard screens, you can always click the "< Back" button to go back to the previous step, or click the "Cancel" button to terminate the creation of a new guest VM.

2. To create a new guest VM, click the "New" button to start the "Create Virtual Machine" wizard (Figure 25.4).
3. Since our guest VM will be an installation of Solaris, type in **`Solaris`**. Click the "Operating System" drop-down menu and select "Solaris". In the "Version" drop-down menu, select "Oracle Solaris(64 bit)". Click the "Next >" button to continue (Figure 25.5).

The base memory size must be changed to at least 1536 MB. If your system has a lot of RAM, use more. In our case, we used 2000 MB. Any number within the green area below the slider is considered a value that should not slow down your computer too much. Click the "Next >" button to continue.

4. This next screen is used to create the virtual hard drive, the amount of disk space on the host OS hardware that will be available to the VM guest. If this is your first VM, keep the default of "Create a virtual hard drive now" and click "Create" to continue.
5. On the next screen, select "VDI (Virtual Disk Image)" and click the "Next" button.
6. You can now choose whether you want "Dynamically allocated" or "Fixed size" storage. The first option uses disk space as needed until it reaches the maximum size that you will set in the next screen. The second option creates a disk the same size as that specified amount of disk space, whether it is used or not. Choose the first option if you are worried about disk space; otherwise, choose the second option as it allows VirtualBox to run slightly faster. Click the "Next >" button to continue.
7. You can now choose to set the size (or upper limit) of the VM. If you plan to install Solaris into the VM, increase the size to at least 100 GB or you will receive an error during the Solaris installation. Whatever size you set, make sure that your computer has enough free disk space to support it. Use the folder icon to browse to a directory on disk with sufficient space to hold your VM.
8. Once you make your selection and press "Next", you will see a summary of your choices. You can use the "Back" button to return to a previous screen if you wish to change any values. Otherwise, click "Create" to close the wizard. Your VM should now show up in the left-hand box of the VirtualBox VM Manager screen, as seen in example in Figure 25.6.
9. Configuring the storage device: On our installation, this step was automatically configured when we wanted to install Solaris from a DVD in a drive already mounted on the system at Serial Advanced Technology Attachment (SATA) port 1.

   You may want to configure it to use your installation media, depending on what that is. Click the storage hyperlink (the word "Storage") in the "Storage"



Solaris
Powered Off

General
Name: Solaris
Operating System: Oracle Solaris 11 (64 bit)

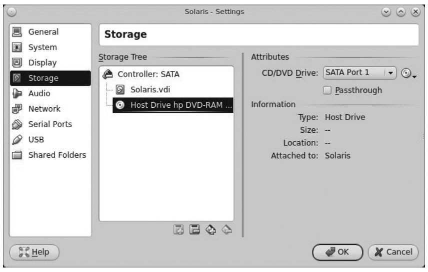
System
Base Memory: 2000 MB
Boot Order: Floppy, CD/DVD, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging

Display
Video Memory: 12 MB
Remote Desktop Server: Disabled
Video Capture: Disabled

Storage
Controller:
SATA
SATA Port 0: Solaris.vdi (Normal, 100.00 GB)
SATA Port 1: [CD/DVD] Empty

Audio
Host Driver: OSS Audio Driver
Controller: ICH AC97

FIGURE 25.6 The new VM for Solaris.



Storage

Storage Tree
- Controller: SATA
  - Solaris.vdi
  - Host Drive hp DVD-RAM ...

Attributes
CD/DVD Drive: SATA Port 1
Passthrough

Information
Type: Host Drive
Size: --
Location: --
Attached to: Solaris

FIGURE 25.7 Storage settings of the VM.

frame of the VirtualBox VM Manager window. This allows you to access the storage screen seen in Figure 25.7. Double-click the word “Empty”, which represents your DVD reader. If you wish to access the Solaris installer from your DVD reader, double-check that the slot is pointing to the correct location (e.g., SATA port 1) and use the drop-down menu to change it if the location is incorrect. Click the “CD/DVD Device” drop-down menu to change it from “Empty” to the correct host drive value.

If you prefer to use an ISO file that is stored on your hard disk, click the DVD icon “-> Choose a virtual CD/DVD disk file” to open a browser menu where you can navigate to the location of the ISO file. Highlight the desired ISO file and click “Open”. The name of the ISO file will now appear in the “Storage Tree” section.

10. You are now ready to install Solaris into your VM. Simply highlight the VM and click on the green “Start” button. A window will open, indicating that the VM is starting. If you have a DVD inserted, you should hear it spin and it should start to boot into the installation program. If it does not or if you are using an ISO file stored on the hard disk, press `<F12>` to select the boot device when you see the message to do so, then press `C` to boot.
11. When the Solaris installation is completed, reboot the Solaris guest (remembering to remove the DVD medium before the reboot takes effect). You should then be able to see a Solaris window showing inside of a PC-BSD screen display, as seen in Figure 25.8.
12. To close down both the VirtualBox Solaris guest and the VirtualBox program itself, first make the Solaris pull-down menu choice “System >Shutdown”. Then, after Solaris has completely shut down, from the VirtualBox Manager window, make the pull-down menu choice “File>Exit”.

Conclusions: Making the proper configuration choices when installing a guest operating into a VM affects the performance of that OS significantly. You must be aware of memory and disk capacities on your host machine to optimize the performance of both host and guest.

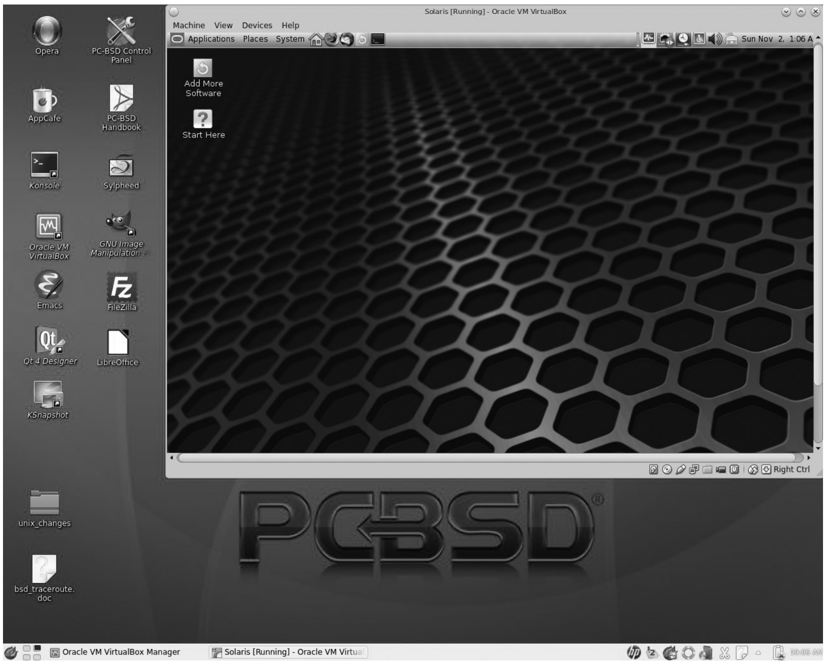


FIGURE 25.8 Solaris guest window in a PC-BSD host display.

25.4.4 Securing an FTP Server in a VirtualBox Guest

The following example shows how to take advantage of having a VM running on your host machine. In it, we secure an FTP server by isolating it in a VirtualBox VM guest. The host in the example is a PC-BSD system with VirtualBox preinstalled, and a Solaris guest OS acting as the secured FTP server on a LAN.

This does not preclude you using Solaris as the host system instead, and installing PC-BSD in VirtualBox as a guest. Or, for that matter, using any supported OS as the host, and installing any other supported OS as the guest on that host. We have chosen to show this methodology with these particular example systems to illustrate the process with two UNIX systems. The particulars of accomplishing the same thing with other OSs are left up to you.

This assumes you have done Section 23.2.5.5, "Examples of System Service Management"; in particular, the first example in that section. It illustrates the enabling of TELNET and FTP servers on PC-BSD and Solaris.

Example 25.8: Managing Services on a VirtualBox Guest

Objectives: To secure an FTP server on Solaris running as a VirtualBox VM guest on a PC-BSD host machine.

Introduction: This method, and the details of the following example, highlight the important use of virtualization-system security. By running an FTP server on a VM that is isolated from the host OS, and is running a completely different kernel than the host system, you create a more secure host.

What we will accomplish in this example is as follows:

- Assign an IP address, within a range of addresses, to the VM guest.
- Check what the actual address assigned to our VM guest is.
- Ensure that the VM guest is attached to a bridged network adapter on the host. What that means is that our host NIC (`bge0`) can now handle two IP addresses: the host IP address and the VM guest IP address.
- Test the FTP server on the VM guest via a LAN.

Prerequisites:

1. You must have completed Example 25.7.
2. You must make sure you have VirtualBox installed on your PC-BSD system, and know how to launch it.
3. You must install Solaris as a VM guest, as shown in Example 25.7.
4. You must enable the FTP services on both systems, as shown in the first example in Section 23.2.5.5.

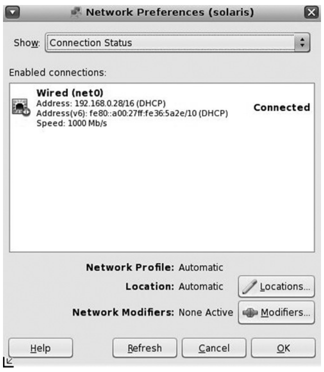
   Make sure the firewall on the PC-BSD host system has the exception to allow FTP traffic incoming on port 21, as shown in that example. Also, make sure you test the FTP connections on the Solaris VM guest as shown in the first example in Section 23.2.5.5 by using the loopback `localhost`.
5. Your host machine must be connected to a LAN and the Internet.

Procedures:

1. To ensure that the guest VM IP address is in the subnetwork of the host, type the following in a PC-BSD host console window as superuser when the Solaris VM guest is not running:

   ```
   # VBoxManage modifyvm "VMname" -natnet1 "192.168/16"
   ```

   where **`"VMname"`** is the name you gave the guest VM when you installed it. This ensures that when the Solaris guest VM is run, the IP address of the NIC will be in the range of 192.168.0.0 to 192.168.254.254. In this example, the guest VM is assigned an IP address of 192.168.0.28, and the host an IP address of 192.168.0.13.
2. Launch VirtualBox on PC-BSD, and start the Solaris VM guest.
3. In the Solaris guest VM window, make the Solaris menu choice “System > Administration > Network”. The “Network Preferences” window opens on screen, with the title “Network Preferences (Solaris)”, as seen in Figure 25.9. Note the virtualized IP address of the Solaris guest, which is seen in this window as something like 192.168.0.28/16 (DHCP) for the first NIC. It was assigned within the range by the VBoxManager command from Step 1.



Network Preferences (solaris)

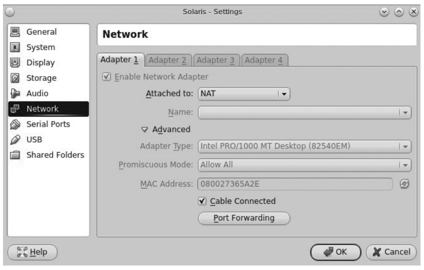
Show: Connection Status

Enabled connections:

Wired (net0)
Address: 192.168.0.28/16 (DHCP)
Address(v6): fe80::a00:27ff:fe36:5a2e/10 (DHCP)
Speed: 1000 Mb/s
Connected

Network Profile: Automatic
Location: Automatic
Network Modifiers: None Active

FIGURE 25.9 VirtualBox network preferences display.

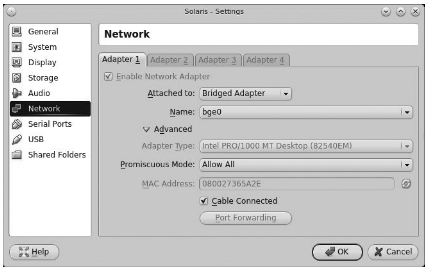


Solaris - Settings

Network

☑ Enable Network Adapter
Attached to: NAT
Name:
Advanced
Adapter Type: Intel PRO/1000 MT Desktop (82540EM)
Promiscuous Mode: Allow All
MAC Address: 080027365A2E
☑ Cable Connected

FIGURE 25.10 VirtualBox network settings window.



Solaris - Settings

Network

☑ Enable Network Adapter
Attached to: Bridged Adapter
Name: bge0
Advanced
Adapter Type: Intel PRO/1000 MT Desktop (82540EM)
Promiscuous Mode: Allow All
MAC Address: 080027365A2E
☑ Cable Connected

FIGURE 25.11 Bridged adapter setting.

In other words, the guest and the host are on the same subnetwork, but will have different IP addresses.

3. From the VirtualBox menus at the top of the VM screen, make the menu choice "Devices > Network > Network Settings". The "Network Settings" window appears on screen, named "Solaris–Settings", as shown in Figure 25.10.
4. Click on the down-facing arrow to the right of "NAT", in the "Attached to:" pull-down menu. This is for Adapter 1 as seen in Figure 25.10.
5. Make the pull-down choice "Bridged Adapter" as the "Attached to:" option, as shown in Figure 25.11.
6. When you have made the "Bridged Adapter" choice, click on the "OK" button at the bottom of the "Network Settings" window.

7. Click the "OK" button to exit from the "Network Settings" window.

   What we have accomplished to this point is:

   - Assigned an IP address, within a range of addresses, to the VM guest.
   - Checked what the actual address assigned to our VM guest is.
   - Ensured that the assigned IP address for the VM guest is attached to a bridged adapter on the host. What that means is that `bge0`, the host NIC, now can handle two IP addresses, the PC-BSD host IP address and the Solaris VM guest IP address.

8. From another machine on the LAN to which your host is connected, FTP to your Solaris VM guest host. In our case, following from the previous steps, we used the command:

```
$ ftp 192.168.0.28
```

With the Solaris VM running, the incoming FTP traffic to 192.168.0.28 will be routed to the Solaris VM guest, and you will be able to use it as a secure FTP server.

Conclusions: Using a bridged adapter on a VM guest allows you to access the guest from either a LAN or the Internet, through the same physical NIC on the host. Using this security technique, you can isolate services running on the guest VM from the host system.

EXERCISES 25.7

Repeat Example 25.8 for the SSH service, to allow you to `ssh` from another system on your network to the Solaris guest running on a PC-BSD host.

EXERCISES 25.8

Do Example 25.8 with Solaris as the host system, and PC-BSD as the guest.

EXERCISES 25.9

Repeat Exercise 25.7 for the TELNET service, to allow you to `telnet` from another system on your network to the PC-BSD guest running on a Solaris host.

25.4.5 Installing PC-BSD or Solaris as a Guest VM on a LINUX or Windows Host

The methods shown in this section show how to install PC-BSD or Solaris as a guest VM on a host machine running either LINUX or Windows. They provide the following advantages:

- Excellent practice for installation of PC-BSD or Solaris in optional ways
- Determination of whether all of your specific hardware is supported by PC-BSD or Solaris
- Allow you to try multiple versions of PC-BSD or Solaris

These methods differ from Section 25.4.3 (Example 25.7), which shows a guest installation using an ISO image burned onto a DVD. The methods in this section use prebuilt, online, and easily downloadable disk images of the entire guest OS. The process of installing an OS from an ISO-created DVD is what we detailed for PC-BSD and Solaris in Chapter 23.

We use Ubuntu LINUX 15.04 and Windows 10 as our host OSs. The methods we apply to those systems in this section can be also applied to OS X.

25.4.5.1 Installing a PC-BSD Guest on a LINUX Host

We describe how to prepare VirtualBox on the host OS for an installation of PC-BSD using the downloadable **.vdi** and **.ova** images obtainable at http://www.pcbsd.org/en/download.html.

We downloaded **PCBSD10.2-RELEASE-x64-desktop.vdi.xz** for LINUX and **PCBSD10.2-RELEASE-x64-desktop.ova** for Windows, before beginning guest installation.

Be aware that these file names may change at the website as later releases of PC-BSD become available as prebuilt VirtualBox guests.

Before attempting this section, you should have installed VirtualBox on your LINUX host machine. The instructions for doing that are available at www.virtualbox.org. At the time of the writing of this book, VirtualBox 5 was available for LINUX systems. The minimum system requirements for installing the premade PC-BSD guest OS are:

64-bit processor architecture

1024 MB base memory size

A virtual disk at least 50 GB in size for a PC-BSD installation

A bridged network adapter

Once you have downloaded and installed VirtualBox on the host LINUX system, use the following steps to build a PC-BSD guest on that host:

1. PC-BSD provides prebuilt VirtualBox disks which create a premade VM guest with PC-BSD already installed as the guest. Download the premade file from the website given. The VirtualBox file ends in a **.vdi.xz** extension. The **.xz** means that the file needs to be decompressed first so that it just ends with a **.vdi** extension.
2. On a LINUX system, such as Ubuntu, to decompress the compressed file, use the **xz** command as shown here, using the name of the file which you downloaded as the command argument:

   ```
   # xz -d -k PCBSD10.2-RELEASE-x64-desktop.vdi.xz
   ```

   This is a large file, so the command will take a few minutes to extract the image. If you do not want to retain the compressed file after decompression, omit the `-k` option. You will be back at the shell prompt when the command has finished the decompression.

3. Once the file is extracted, run VirtualBox with the command **`sudo VirtualBox`**.
4. When the VirtualBox VMM appears on screen, use Steps 1–4 in Example 25.7 to create a new VM, using the "New" icon, with the following characteristics:
   a. OS type "BSD" and FreeBSD version "64-bit"
   b. A minimum of 128 MB of main memory. A more realistic value for main memory specification would be 2 GB, if you want to achieve more than minimal performance. This value also depends on how much main memory your host system has installed.

   When you get to Step 4 of Example 25.7, make the selection "Use an existing virtual hard drive file", instead of the selection "Create a virtual hard drive now".
5. Use the browse icon in the lower-right corner of the VirtualBox "Create Virtual Machine Wizard" box to browse to the location of the **.vdi** file you extracted in Step 2. Highlight that downloaded file in the browser window, then press "Open". Click the "Create" button to finish the wizard. You will be returned to the VirtualBox VMM window.
6. Start the new VM guest from within the Virtualbox VMM window on the host by highlighting the new machine and selecting the "Start" icon. The VM guest will boot into the postinstallation configuration screens so that the system can be configured, as shown in Section 23.2.3. Once the display wizard is finished, and the login menu appears, input the username and password that you configured at the "Create a User" Screen. You will be logged into the new PC-BSD guest machine.
7. Use PC-BSD. When ready to shut the PC-BSD system down, use the techniques shown in Section 23.2.5.3, "Graceful Shutdown." When PC-BSD completely shuts down, you are returned to the VirtualBox VMM on the host.
8. To delete the PC-BSD guest VM, make the VirtualBox VMM menu choice "Machine>Remove" when the PC-BSD guest VM is highlighted. In the dialog box that appears, if you make the choice "Delete all files", be aware that the **.vdi** file you extracted in Step 2 will also be deleted!

*25.4.5.2 Installing a PC-BSD Guest Using an **.ova** File on a Windows Host*

A file that ends in an **.ova** extension is a tarball of a VM that follows the open virtualization format (OVF). This file can be used in any virtualization technology that supports OVF, such as VirtualBox.

Once you have downloaded and installed VirtualBox on the host Windows system, use the following steps to build a PC-BSD guest on that host:

1. PC-BSD provides prebuilt VirtualBox disks which create a premade VM guest with PC-BSD already installed as the guest. Download the premade **.ova** file from the

website given. On our Windows system, we downloaded **PCBSD10.2-RELEASE-x64-desktop.ova**.

2. If you double-click the **.ova** file on a Windows system, it will automatically open the image for you in the default virtualization application, which we assume here is VirtualBox.
3. The first time a PC-BSD **.ova** file is opened on a Windows system, a screen will open so that you can review the VM's settings that came with the file.

Depending on the settings, you can either type in the desired value or select it from a drop-down menu. Once you are finished, click the "Import" button. It will take a few minutes for the import to complete and a status bar will indicate the status of the import. Once imported, the VM will show in the left-hand frame of VirtualBox. Highlight the VM and click "Start" to boot into the image.

4. When using the "desktop" edition download of PC-BSD, the VM will boot into the postinstallation configuration screens so that the system can be configured, as shown in Section 23.2.3. Once the display wizard is finished and the login menu appears, input the username and password that you configured at the "Create a User" screen.
5. Use PC-BSD. When ready to shut the PC-BSD system down, use the techniques shown in Section 23.2.5.3, "Graceful Shutdown." When PC-BSD completely shuts down, you are returned to the VirtualBox VMM.
6. To delete the PC-BSD guest VM, make the VirtualBox VMM menu choice "Machine>Remove" when the PC-BSD guest VM is highlighted. In the dialog box that appears, if you make the choice "delete all files", be aware that the **.ova** file is retained!

EXERCISE 25.10

Use the methods shown in Section 25.4.5.2 to install a PC-BSD guest on a Windows or LINUX system of your choice.

25.4.5.3 Installation of a Solaris Guest on LINUX and Windows Hosts

This section shows the procedure for installing a Solaris guest VM in a LINUX or Windows host running VirtualBox. The techniques shown here work for both LINUX and Windows systems.

The minimum requirements on the LINUX or Windows host are:

X86 system with a minimum of 4 GB RAM.

A minimum of 4 GB free disk space is needed for initial installation.

The available space on the virtual disk can grow to a maximum of 64 GB.

VirtualBox is already installed.

Follow the steps listed here to install, use, and remove the Solaris guest VM:

1. Download a VM "template" for Solaris from the following website or its latest equivalent:

   http://www.oracle.com/technetwork/server-storage/solaris11/downloads/vm-templates-2245495.html

   The file you select to download will be a VM template, as an **.ova** file, for the latest version of Solaris.

2. Start the VirtualBox application on the LINUX or Windows host.
3. From the file menu in the VirtualBox VMM, choose "Import >Appliance".
4. Select the **.ova** file from the directory to which it was downloaded in Step 1.
5. In the "Appliance Import Wizard" window that opens, change "Appliance Import" Settings to suit your system. For example:

   RAM: Systems with more RAM should increase this amount (up to half of available RAM is recommended).

   Virtual Disk Image: Select a directory in which the image size can expand (up to 64GB).

6. Once imported, the virtual machine will show in the left frame of the VirtualBox VMM. Highlight the VM and click "Start" to boot into the image.

   The VM will boot into text-based postinstallation configuration screens, so that the system can be configured. Proceed through configuration screens that prompt for host name, time zone, default user and password, and root password. These are described in graphical format in Section 23.2.4. Then login.

7. Use Solaris. When ready to shut the Solaris system down, use the techniques shown in Section 23.2.5.3, "Graceful Shutdown." When Solaris completely shuts down, you are returned to the VirtualBox VMM.
8. To delete the Solaris guest VM, make the VirtualBox VMM menu choice "Machine>Remove" when the Solaris guest VM is highlighted. In the dialog box that appears, if you make the choice "delete all files", be aware that the **.ova** file is retained!

EXERCISE 25.11

Use the methods shown in Section 25.4.5.3 to install a Solaris guest on a Windows or LINUX system of your choice.

SUMMARY

In this chapter, we illustrated what a VM environment for our UNIX OSs can be. We showed three popular and important facilities for creating a VM environment: PC-BSD `iocage`-managed jails, Solaris zones, and VirtualBox.

We illustrated two modern virtualization techniques, full virtualization and OS-level virtualization, with three example facilities. What differentiates these three facilities is that for the first two (OS-level virtualization), all virtual environments are running under the same kernel. In the third one (full virtualization), any number of different kernels can be running simultaneously on the hardware of one computer.

But it is important to note that, for the three facilities we show, the OS(s) all use the same underlying ISA of the hardware processor(s). In our cases, this is the X86 architecture, as opposed to SPARC or ARM architectures.

For each facility, we used by-example descriptions of how to install it as a VM. We also showed an instance of how to put an application into that VM. For each facility, we gave reference material on the commands and operations used in the facility.

A very important application of these methodologies is to provide a measure of system security.

QUESTIONS AND PROBLEMS

1. Create a new jail in PC-BSD with `iocage`. Then, install packages using the `pkg` command in that jail that you find useful, or with which you would like to experiment.

2. Following the methods of Section 23.2.5.5 (the first example in that section), create and enable an FTP server inside of a VirtualBox VM that has a PC-BSD guest OS installed. Then place sample files in the default account home directory you have created on the guest. Test the FTP login to the guest, from your local host, from a LAN, and from the Internet. Also, use FTP commands to download the sample files you put in the home directory to another computer on your LAN, and to another computer over the Internet.

3. Repeat Problem 23.2 inside of a PC-BSD jail. This allows you to create an anonymous FTP server inside of a secure environment. Then, place sample files in the account home directory, **/var/ftp**. Test the anonymous FTP login, both from your local host, from a LAN, and from the Internet. Also, use FTP commands to download the sample files you put in **/var/ftp** to another computer on your LAN, and to another computer over the Internet.

4. Create a new zone in Solaris using the same methods shown in Example 25.5. Then install packages in this new zone using the `pkg` command that you find useful, or with which you would like to experiment.

5. Install VirtualBox onto a computer running LINUX, and then install PC-BSD TrueOS server in a VM guest on that computer.

6. In a running VM guest, from the VirtualBox Devices pull-down menu, enable the bidirectional shared clipboard feature. Then, copy text by using the mouse in the VM guest OS, and paste it into an application running on the host system.

7. Of the three virtualization methodologies shown in this chapter, which is the most useful for you, given the host OS you have on your computer? For example, if you are

running PC-BSD, and have installed a ports jail, what kinds of application packages would you run in the jail? Why would you be running them in a jail?

8. Using the methods of this chapter, Section 17.5.7 on Git, Section 23.9.3 on PC-BSD access control lists (ACLs), and Section 24.5 on Solaris ACLs:

   a. Create a new PC-BSD Jail or Solaris zone that has access via SSH to and from the Internet.

   b. In that jail or zone, create a project directory for a Git repository that a user base of remote users can push to and pull from.

   c. Use the appropriate ACL commands to set the ACLs on that project directory so that local and remote users can access it, and can interact with the Git repository in that project directory.

   d. Design the directory structure of the repository for the intended local and remote user base.

   e. Create the Git repository in the project directory according to your design.

   f. Test the Git repository, both locally and from the Internet, to confirm that local allowed users and the remote user base can work with the repository to push to and pull from it.

   g. Test that unallowed local users or Internet traffic cannot interact with the repository.

Glossary

absolute pathname: A pathname that starts with the root directory.

access privileges: The type of operations that a user can perform on a file. In UNIX, access rights for a file can be read, write, and execute.

access time: The time taken to access a main memory location for reading or writing.

ACE (access control entries): Define access permissions for a particular class of user to objects such as files. The list of ACEs is numbered, starting from zero. The position of an ACE within an ACL is called an *index*.

ACL (access control list): A finer-grained discretionary permission control mechanism on objects such as files. Contrast with traditional UNIX permissions model.

aclinherit: A ZFS command variable that controls the behavior of ACL inheritance, when used with the applicable modifying command, on a designated ZFS dataset.

aclmode: A ZFS command variable that controls the behavior of initial or modified ACL application to a designated ZFS dataset.

active socket: A socket that serves a client's request(s). For connected-oriented communication, it is created by the `accept()` system call. An active socket has the life span of a connection between a client and the server process. For this reason, an active socket is also known as an **ephemeral socket**.

adaptive Lempel-Ziv coding: The most widely used lossless compression scheme for encoding variable-length block of characters a fixed-length block of bits. In this compression scheme we assume that characters occur independently and with known probabilities, and that the probabilities are the same for all positions.

address bus: A set of parallel wires that are used to carry the address of a storage location in the main memory that is to be read or written.

address space: See **process address space**.

alias: See **pseudonym**.

application programmer's interface (API): The language libraries and system call layer form the application programmer's interface.

application software: Programs that we use to perform various tasks on the computer system, such as word processing, graphing, picture processing, and Web browsing.

application user's interface (AUI): The application software that a user can use forms the application user's interface.

archive: A collection of files contained in a single file in a certain format.

array: A named collection of items of the same type stored in contiguous memory locations.

array indexing: The method used to refer to an array item by using its number. The items in an array are numbered, with the first item numbered 1 (in some languages such as C the first item is numbered 0).

assembler: A program that takes a program in assembly language and translates it into object code.

assembly language: See **low-level programming language**.

assignment statement: A shell command that is used to assign values to one or more shell variables.

asynchronous I/O: The I/O based on nonblocking I/O calls, for example, through nonblocking `read()` and `write()` system calls. It may also be performed using the `select()` system call.

atomic write: The amount of data a writer process can write to a file, pipe, or bounded buffer without interruption.

attributes: The characteristics of a process (or file) such as the name of the owner of the process and process size.

background process: When a process executes such that its standard input is not connected to the keyboard, it is said to execute in background. The shell prompt is returned to the user before a background process starts execution, thus allowing the user to use the system (i.e., run commands) while the background processes execute.

bash: The abbreviation for Bourne Again shell.

batch operating system: An operating system that does not allow you to interact with your processes. The VMS system has a batch interface. UNIX and LINUX also allow programs to be executed in the batch mode, with programs executing in the background.

big endian byte order: A storage (or transmission) order in which the low byte of a multiple-byte data item (char, int, long, etc.) is stored in the high byte of the memory and the high byte of the data is stored in the low byte of memory.

BIOS (Basic Input/Output System): Usually resident in ROM hardware, and deployed after POST performs some system hardware integrity checks, BIOS then searches, loads, and executes the boot loader program GRUB.

bistate devices: The devices, such as transistors, that operate in "on" or "off" mode.

bit: Stands for binary digit, which can be 0 or 1. It is also the smallest unit of storage and transmission.

bit mask: A sequence of bits (usually a byte or multiple bytes) used to retain values of certain bits in another byte (or multiple bytes), or to set them to 0s or 1s, by using a logical operation such as AND or OR.

blind carbon copying e-mail: Sending a copy of an e-mail message composed by you to someone other than the intended recipient. Who received the carbon copy of a message is not known to the receiver because this information does not appear in the e-mail header.

block special files: The UNIX files that correspond to block-oriented devices (see **block-oriented devices**). These files are located in the **/dev** directory.

blocking input: An input (i.e., read) operation that blocks if the file it is reading from is empty.

blocking output: An output (i.e., write) operation that blocks if the file it is writing to is full.

block-oriented devices: The devices, such as disk drives, that perform I/O in terms of blocks of data (e.g., in 512-byte chunks).

blocks: See **disk blocks**.

boot environment: A boot environment is a bootable environment consisting of a ZFS root file system and, optionally, other file systems mounted underneath it. Exactly one boot environment can be active at a time.

bounded buffer: A fixed-size buffer that is typically used as a circular buffer.

bounded-buffer reader–writer problem: A synchronization problem in which reader and writer processes run forever and communicate (i.e., read and write) through bounded buffer. See **bounded buffer**.

break point: A program statement where the execution of the program stops while using a symbolic debugger.

bsd socket: See **socket**.

bss: The uninitialized data area in the memory image of a UNIX process

byte: In contemporary literature, a byte refers to eight bits. For example, 10101100 is a byte. In not-so-recent literature, the term also used to refer to nine bits. Because of this, a byte is also called an *octet*. A storage location that can store eight bits is also known as a byte.

C preprocessor: A program that takes a C program as input and processes all the statements that start with the # sign. It produces output that is taken by the C compiler as input to produce the assembly code. A typical C compiler performs all the tasks necessary to produce the executable code for a C program. These tasks are preprocessing, compilation, assembly, and linking.

carbon copying e-mail: Sending a copy of an e-mail message composed by you to someone other than the intended recipient. Who received the carbon copy of a message is known to the receiver because this information appears in the e-mail header.

central processing unit (CPU): Also known as the *brain* of a computer system, the CPU executes a program by reading the program instructions from the main memory. It also interacts with the I/O devices in the computer system.

character special files: The UNIX files that correspond to character-oriented devices. These files are located in the **/dev** directory.

character user interface (CUI): See **commandline user interface (CUI)**.

character-oriented devices: The devices, such as keyboards, that perform I/O in terms of one byte at a time.

checksum: A 256-bit hash of the data in a file system block. The checksum capability can range from the simple and fast fletcher4 (the default) to cryptographically strong hashes such as SHA256.

child process: A process created on behalf of another process. In UNIX, the fork system call has to be used to create a child process. The child process is an exact copy of the process executing fork (see **parent process**).

chroot: An operation that changes the apparent root directory for the current running process and its children. A program that is run in such a modified environment cannot name (and, therefore, normally cannot access) files outside the designated directory tree. The term "chroot" may refer to the `chroot` system call or the chroot wrapper program. The modified environment is called a *chroot jail*. Only the root user can perform a chroot.

class: A template for creating user-defined objects. Class definitions normally contain method definitions which operate on instances of the class.

client software: In the client–server software model, the client software, when executed, takes the user commands and sends them as requests to the server process. The server process computes the responses for requests and sends them to the client, who handles them according to the semantics of the command. All Internet applications are based on the client–server model of computing.

clock tick: A clock in a computer system ticks as frequently as dictated by the frequency of the clock (ticks per second). For a system clock that is dependent on the frequency of the power-line signals (50 or 60 per second), it ticks every 1/50 (or 1/60) of a second.

cluster: The minimum unit of disk storage, which is one or more sectors.

coding rules: A set of rules used by programmers for writing programs. Such rules are usually designed to enhance the readability of programs and to keep consistency in the "look" of the source programs produced by an organization or a coding team. The use of coding rules helps a great deal during the maintenance phase of a software product.

colon hexadecimal notation: A notation used to write 128-bit IPV6 addresses in a compact notation, in which each 16-bit chunk is represented in hexadecimal separated by colons, as in 76F4:9D5F:FFFF:FFFF:0:3276:70BD:FFFF.

command grouping: Specifying two or more commands in such a manner that the shell executes them all as one process.

command interpreter: A program that starts running after you log on to allow you to type commands that it tries to interpret and run. In UNIX, the command interpreter is also known as a **shell**.

command line: A line that comprises a command with its arguments and is typed at a shell prompt. You must hit the `<Enter>` key before the command is executed by a shell.

command line arguments: The arguments that a command needs for its proper execution, which are specified in the command line. For example, in the command `cp f1 f2` , `f1` and `f2` are command line arguments. Within a shell script, you can refer to these arguments by using positional parameters $1–$9.

command mode operation: Operation that consists of key sequences that are commands to a text editor for taking a certain action.

command substitution: A shell feature that allows the substitution of a command by its output. To do so, you enclose a command in back quotes (grave accents). Thus, in the echo `'date'` command, the output of the `date` command substitutes for the `date` command, which is then displayed on the display screen.

commandline user interface (CUI): If you use a keyboard to issue commands to a computer's operating system, the computer is said to have a commandline user interface.

comments: Short notes placed in the program's source code that explain segments of the code. Comments must be distinguished so they are not executed as program commands (statements). For shell scripts, a comment line must start with the # sign.

commit (as a noun): A single point in the Git history; the entire history of a project is represented as a set of interrelated commits. The word "commit" is synonymous with the words "revision" or "version". Also synonymous with a commit object.

commit (as a verb): The action of storing a new snapshot of the project's state in the Git history, by creating a new commit representing the current state of the index and advancing HEAD to point at the new commit.

compatibility release: A release/version of a software that is meant to provide uniformity between any particular implementation and its perceived competitors.

compiler: A program that takes a program written in a high-level language and translates it into the corresponding assembly program. Almost all C and C++ compilers also perform the tasks of preprocessing, assembly, and linking.

computer network: An interconnection of two or more computing devices. A device on a network is commonly called a *host*.

concurrent client: A client that handles multiple descriptors, such as an FTP or SSH client.

concurrent server: A server process that, on receiving a client request, creates a child process, delegates the rest of communication with the client to that child process, and goes back to wait for another client request.

configuration file: A file that contains the definitions of various environment variables to set up your environment while you use a shell. Every shell has a start-up configuration file for every shell in your home directory that is executed when that shell starts running (e.g., **.cshrc** for the C shell).

connection: A connection without an explicit request from the client.

control bus: A set of parallel wires that are used to carry control information from the CPU to the main memory or an I/O device. For example, it carries the "read" or "write" instruction from the CPU to the main memory.

control unit: The part of a CPU that interacts with the devices in a computer system (memory, disk, display screen, etc.) via controllers (see **controllers**) in these devices. It also fetches a program instruction from the main memory, decodes it to determine whether the instruction is valid, and then passes it on to the execution unit (see **execution unit**) for its execution.

controllers: The electronic part of an I/O device, which communicates with the CPU or other devices.

CPU scheduling: A mechanism that is used to multiplex the CPU among several processes. This results in all processes making progress in a fair manner and increased utilization of hardware resources in the computer system.

CPU state: The values of the CPU registers at any given time, including the value of the program counter.

CPU usage: The percentage of the time the CPU in a computer system has been used since the system has been up.

critical section: A piece of code in a thread in a cooperating/concurrent process that accesses shared data.

critical section problem: Writing code of cooperating/concurrent processes to ensure serial execution of critical sections in these processes.

cryptography: The science of transforming information so that it is unintelligible to the inexperienced and understandable to those who have some special knowledge, known as the *decryption key*.

csh: The abbreviation of C shell.

current directory: The directory that you are in at a given time while using a computer system. In UNIX, the `pwd` command can be used to display the absolute pathname of your current directory.

current job: The job (process) that is currently being executed by the CPU.

cursor: The point that tells you at which part of the screen you are located at a given time.

daemon: A system process executing in the background to provide a service such as printing. For example, in a typical UNIX system the lpd daemon offers the printing service and fingerd offers the finger service. A process not connected to standard input or output.

DAG (directed acyclic graph): Refers to commit objects that, in pictorial form, show parents/descendants (directed).This picture of commit objects is acyclic (there is no chain which begins and ends with the same object).

data bus: A set of parallel wires that are used to carry data from the CPU to a subsystem (memory or I/O device), and vice versa.

dataset: A generic name for the following ZFS components**:** Clones, file systems, snapshots, and volumes. Each dataset is identified by a unique name in the ZFS name space. Datasets are identified using the following format: `Pool/path[@snapshot]`

decryption: The process of converting an encrypted file (see **encryption**) to its original version.

deduplication: Data deduplication is a method of reducing storage capacity needs by eliminating redundant data. Only one unique instance of the data is retained on storage media. Redundant data is replaced with a pointer to the unique data copy.

descriptor set: A bit mask in which a bit represents a descriptor and the value of a bit indicates the state of the corresponding descriptor; that is, ready or not ready for an I/O operation or exception handling.

desktop manager: A software system that provides a graphical method of interacting with the operating system.

disk blocks: The unit of disk I/O. It is one or more sectors (512 bytes).

disk scheduling: In a time-sharing system, several requests can come to the operating system for reading or writing files on a disk. The disk scheduling code in the operating system decides which request should be served first.

dispatcher: Operating system code that takes the CPU away from the current process and gives it to the newly scheduled process (i.e., it saves the state of the current process and loads the state of the newly scheduled process).

domain name system (DNS): A distributed database that can be used to convert the domain name of a host to its IP address.

dot file: See **hidden file**.

dotted decimal notation (DDN): 32-bit (4-byte) IP addresses are difficult to remember. This notation is used to express every byte of an IP address in equivalent decimal and place dots between them. Thus, the IP address 11000000100011000000101000 000001 (in binary) is 192.140.10.1 in the dotted decimal notation.

dynamic analysis: The analysis of a program as it executes. The analysis comprises debugging, tracing, and performance monitoring of the program, including testing it against product requirements.

dynamic linking: Linking carried out at runtime.

editor buffer: Usually nonpersistent memory locations open during a text editor session that store information on the state of information being edited.

encrypted file: A file that contains a file's contents after it has gone through the encryption process (see **encryption**).

encryption: The process of converting a file's contents to a completely different form by using a process that is reversible, thereby allowing recovery of the original file.

end-of-file (eof) marker: Every operating system puts a marker at the end of a file, called the end-of-file (eof) marker.

environment variables: The shell variables (see **shell variable**) whose values control your environment while you use the system. For example, it dictates which shell process starts running and what directory you are put into when you log on.

ephemeral socket: See **active socket**.

Ethernet: The most famous protocol for physically connecting hosts on local area networks.

Ethernet broadcast address: The all 1s Ethernet address.

execute permission: A UNIX access privilege that must be set for a file to be executed by using the file name as a command. When set for a directory, it allows the directory to be searched.

execution unit: Also called the *arithmetic and logic unit (ALU)*, it executes instructions in a program delivered to it by the control unit.

exit status: A value returned by a process, indicating whether it exited successfully or unsuccessfully. In UNIX, a process returns a status of zero on success and a non-zero value on failure.

expression: A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators, or function calls which all return a value.

external command: A shell command for which the service code is in a file and not part of the shell process. When a user runs an external command, the code in a corresponding file must be executed by the shell. The file may contain binary code or a shell script.

external signal: A signal whose source is not the CPU. For example, pressing `<Ctrl+C>` on the keyboard sends an external signal, also called *keyboard interrupt*, to the process running in the foreground.

FCFS: See **first-come-first-served mechanism**.

Fibonacci series: A series of positive integers with the first two numbers being 0 and 1, and the next number in the series being calculated by adding the previous two numbers. Thus, the first 10 elements of the series are 0, 1, 1, 2, 3, 5, 8, 13, 21, and 34.

FIFO: First-in-first-out order.

file compression: The process of shrinking the size of a file.

file descriptor: A small positive integer associated with every open file in UNIX. It is used by the kernel to access the inode for an open file and determine its attributes, such as the file's location on the disk.

file descriptor: An object of type "int" returned when a file is opened using the system call interface. Subsequent I/O tales place using this int. Used as an index for the per-process file descriptor table (PPFDT). See **per-process file descriptor table**.

file descriptor table: A per-process table maintained by the UNIX system that is indexed by using a file descriptor to eventually access the file's inode.

file handle: A term used for file pointer or file descriptor.

file maintenance: The operation of organizing your files according to some logical scheme.

file pointer: A pointer to an object file returned when a file is opened using the standard I/O interface. Subsequent I/O takes place using this object.

file system: A directory hierarchy with its own root stored on a disk or disk partition, mounted under (glued to) a directory. The files and directories in the file system are accessed through the directory under which they are mounted.

file-system-persistent object: An object that remains in existence for the life of the file system under which it was created.

file system structure: The structure that shows how files and directories in a computer system are organized. On most contemporary systems, the files and directories are organized in a hierarchical (tree-like) fashion.

file table: Also known as the *system-wide file table (SFT)*, a table maintained by the UNIX operating system to keeps track of all the files open in a UNIX system at any given time, which links the PPFDTs and the inode table.

file transfer protocol: An application-level protocol in the TCP/IP protocol suite that allows you to transfer file(s) from a remote host to your host, or vice versa. The UNIX `ftp` command can be used to access this Internet service.

filter: A UNIX term for a command that reads input from standard input, processes it in some fashion, and sends it to standard output. Examples of UNIX filters are `sort`, `pr`, and `tr`.

firewall: A security measure, instituted in PC-BSD and Solaris at the software level, to control access to the system from a network connection.

first-come-first-served mechanism: A scheme that allows print requests (or any other kinds of requests) on the basis of their arrival time, serving the first request first.

focus policy: In the X Window System and XFree86, the way in which the current position of the cursor is made to appear in the current open window, or the relationship of the current position of the cursor and the current active window.

folder: Also known as a *directory*; a place on the disk that contains files and other folders arranged in some organized and logical fashion.

foreground process: A process that keeps control of the keyboard when it executes; that is, the process whose standard input is attached to the keyboard. Only one foreground process can run on a system at a given time.

forwarding e-mail: Sending a copy of an e-mail message received from someone to another e-mail address.

FTP: See **file transfer protocol**.

full association: A full association has been established between two sockets when they both have names bound to them and know the names of each other. Such sockets are known as *fully connected*.

full-screen display editor: An editor that displays a portion of the file being edited in the console window or terminal screen.

full-screen e-mail display systems: E-mail systems that allow you to edit any text you see on a single screen display, as you would on a word processor.

fully parameterized client: A client software that has the flexibility of allowing identification of a particular port number where a server runs. TELNET is an example of a fully parameterized client, because, although the TELNET server normally runs on the well-known port 23, you can run a TELNET server on another port and connect to it by specifying the port number as a command line parameter with the `telnet` command. For example, in the `telnet foo.foobar.org 5045` command, the TELNET client will try to connect to the server running on port 5045.

fully qualified domain name (FQDN): The name of a host that includes the host name and the network domain on which it is connected. For example, www.up.edu is the FQDN for the host whose name is up.edu.

function: A series of commands that are given a name. The commands in a function are executed when the function is invoked (called).

function body: The series of commands in a function.

gateway: See **router**.

general purpose buffer: An area in the main memory maintained by an editor; it contains the most recent cut/copied text.

getty process: At system bootup time, the UNIX system starts running a process on each working terminal attached to the system. This process runs in superuser mode and sets terminal attributes such as baud rate as specified in the **/etc/termcap** file. Finally, it displays the `login:` Prompt and waits for a user to log on.

Git: A UNIX tool for version control and software development sharing, both command-line- and Web-based via Github.

Git branch: A name for a line of commits, also called a *reference*. It shows the parents/descendants of a commit, and thus the typical notion of a "branch of project development."

Git staging model: Git has three main states that files can be in: Modified, staged, and committed. The files are either in the working directory, the index, or the object store.

global variable: A variable that can be accessed by children of the process (executing shell script) in which it is defined.

graphical user interface (GUI): If you use a point-and-click device, such as a mouse, to issue commands to its operating system, a computer is said to have a graphical user interface.

group: In UNIX, every user of the computer system belongs to a collection of users known as the user's group.

GRUB (grand unified bootloader): A bootloader program that loads and executes the kernel and initrd images.

half association: A socket is said to be half associated when a name has been bound to it. Such sockets are said to be half connected.

hard coding: Making a value part of a program as opposed to taking it from an outside source such as the keyboard or a file.

hard link: A mechanism that allows file sharing by creating a directory entry in a directory to allow access to a file (or directory) via the directory. Loosely applied, it is a "pointer" to the inode of a file to be accessed via multiple pathnames. The `ln` command is used to create a hard link to a file.

header: See **program header**.

header file: A file that contains definitions and/or declarations of various items (e.g., constants, variables, and function prototypes) to be used in the program in the C, C++, or Java programming language.

heap: The area in the memory image of a UNIX process used for dynamic memory allocation/deallocation.

here document: A Bourne and C shell feature that allows you to redirect standard input of a command in a script and attach it to data in the script.

hidden file: A file whose name starts with a dot (`.`). Such files are not listed when in the output of the `ls` command unless you use the `ls -a` command. Examples of hidden files are **~/.bashrc**, **~/.cshrc**, **~/.login**, and **~/.profile**.

high-level programming languages: Programming languages such as C, C++, Python, Java, FORTRAN, and LISP that are closer to spoken languages and are independent of the CPU used in the computer system.

hole: An unused area in a file, created with the file pointer, is set, and data is written beyond the current end-of-file.

home directory: See **login directory**.

home page: The contents of a file displayed on the screen (the actual contents can be multiple screens long) for an Internet site.

host: A hardware resource, usually a computer system, on a network.

Huffman coding: A lossless compression scheme for encoding a fixed-length block of characters into a variable-length block of bits. In this compression scheme, we assume that characters occur independently and with known probabilities, and that the probabilities are the same for all positions. If statistics about fixed block of characters are known *a priori*, Huffman coding results in optimal codes.

hypervisor: See **virtual machine monitor**.

I/O-bound process: A process that spends most of its time performing I/O operations, as opposed to performing some calculations by using the CPU.

i-list: A list (array) of inodes on the disk in a UNIX system. See **inode**.

immutable object: An object with a fixed value. Immutable objects include numbers, strings, and tuples. Such an object cannot be altered. A new object has to be created if a different value has to be stored; for example, a key in a dictionary.

index number: See **inode number**.

index screen: The user interface in full-screen-display e-mail systems. It usually consists of three areas: One that contains the message number, sender's e-mail address, date received, size of the message in bytes, and subject line; a second that contains a list of possible commands; and third, a command area where your typed commands are displayed. These screens vary from one system to another.

indexed buffer: A buffer used by a text editor that allows you to store more than one temporary string.

infinite loop: See **nonterminating loop**.

information hiding: A technique used to implement software when the internal structure of a data item is not important. What is important is the types of operations that can be performed on the data and the input/output characteristics of the operations.

init process: The first user process that is created when you boot up the UNIX system.The BSD-style `init` program brings up the system by running the **/etc/rc** script. The rc script is a very easy to understand and manage facility. On a Solaris system, the SMF service starter replaces `init`.

initrd (initial RAM disk): Used by the kernel as temporary root file system until kernel is booted and the real root file system is mounted. It also contains necessary drivers compiled inside, which helps it to access the hard drive partitions, and other hardware.

inode: An element of an array on disk (called the **i-list**) allocated to every unique file at the time it is created. It contains file attributes such as file size (in bytes). When a file is opened for an operation (e.g., read), the file's inode is copied from disk to a slot in a table kept in the main memory, called the inode table (see **inode table**), so that the file's attributes can be accessed quickly.

inode number: A 2-byte index value for the i-list (or inode table) used to access the inode for a file.

inode table: A table (array) of inodes in the main memory that keeps inodes for all open files. The inode number for a file is used to index this array in order to access the attributes of an open file.

insert mode of operation: Mode that allows you to input text to be inserted in the document being edited.

instruction set: The language that a CPU understands. A CPU can understand instructions only in its own instruction set, which is usually a superset of its predecessors made by the same company.

interactive operating system: An operating system that allows you to interact with your processes. Almost all contemporary operating systems, such as LINUX, UNIX, and Windows, are interactive.

internal (built-in) command: A shell command for which the service code is part of the shell process.

internal signal (trap): An interrupt generated by the CPU. This may be caused, for example, when a process tries to access a memory location that it is not allowed to access (see **process address space**).

Internet: See **internetwork**.

Internet domain name system: A distributed database of domain name and IP address mappings. It is maintained by hosts called *name servers*. Every site on the Internet must have at least one computer that acts as a name server.

Internet domain socket: A socket (an interprocess communication endpoint in BSD-compliant UNIX systems) that can be used for communication between processes on the same computer or between different computers on a network or an internet.

Internet login: Logging on to a computer on the Internet.

Internet message access protocol (IMAP): A method of accessing e-mail or bulletin-board messages at a mail server by using a client software, without transferring any files or messages between the two computers.

Internet Protocol (IP): The network layer protocol in the TCP/IP protocol suite that routes packets (known as *datagrams* in TCP/IP terminology) from the source host to the destination host.

Internet service provider (ISP): A company that offers Internet services such as e-mail and Web browsing through dialup or cable connections.

internetwork: A network of computer networks. The ubiquitous internet is called the **Internet**.

internetworking: Making a network of networks. In terms of software, the term internetworking is usually used to refer to writing client–server programs that allow processes on various hosts on the Internet to communicate with each other.

interpreted program: A program that is executed one command (statement) at a time by the interpreter.

interpreter: A program that executes statements (or commands) in a program one by one. An example of an interpreter is a UNIX shell that reads commands from a keyboard or a shell script and executes them one by one.

interprocess communication (IPC) mechanisms: Facilities (channels and operations on them) provided by an operating system that allow processes to communicate with each other. UNIX has several channels for IPC including pipes, FIFOs, and BSD sockets. These channels are created by using UNIX systems called `pipe`, `mkfifo` (`mknod` in older systems), and `socket`.

interrupt: A “signal” that a peripheral hardware device sends to the CPU in order to get its attention.

interrupt service routine: The kernel code to service an interrupt.

interrupt-driven interaction: A mechanism used in modern computer systems in which applications wait for a signal from a particular input device and then take an appropriate action.

intranet: A network of computer networks in an organization that is accessible to people in the organization only.

intranet login: Logging on to a computer on an intranet.

IP address: A 32-bit positive integer (on IPV4) to uniquely identify a host on the Internet. On IPV6, it is a 128-bit positive integer.

IP broadcast address: An IP broadcast address (also known as *broadcast address*) is used by the IP layer in a host to send a datagram to all the hosts on a subnet or to a remote network.

iteration: A single execution of the piece of code in a loop (see **loop**).

iterative server: A server process that, on receiving a request from a client process, prepares a response, sends the response to the client process, and waits for the next request.

job: A print request or a process running in the background.

job ID: A number assigned to a print job. On some systems, it is preceded by the name of the printer.

job number: A small integer number assigned to a background process.

kernel-level threads: The threads created by user programs using thread libraries implemented in the kernel. The kernel handles both processes and threads, including their scheduling.

kernel on intel/AMD: Platforms; the kernel file starts with a 512-byte boot block, then a secondary boot loader block, and then the compressed kernel image. The kernel does all the kernel-space work: Interacting directly with hardware, managing running processes by allocating memory and CPU time, and enforcing access control through ownership and permissions.

keyboard interrupt: An event generated when you press <`Ctrl-C`> that causes the termination of the foreground process.

keyboard macro: A collection of keystrokes that can be recorded and then accessed at any time. This capability allows you to define repetitive multiple keystroke operations as a single command and then execute that command at any time—as many times as you want.

keystroke command: A command that corresponds to pressing one or more keys.

kill ring: Text held in a buffer by killing it and then restored to the document at the desired position by yanking it.

ksh: Abbreviation of Korn shell.

lambda: An anonymous inline function consisting of a single expression which is evaluated when the function is called. The Python syntax to create a lambda function is: `Lambda [arguments]: Expression`.

language libraries: A set of prewritten and tested functions for various languages that can be used by application programmers instead of having to write their own.

last line mode (command mode): A state that the vi and vim editors can be in that allows entry of editor commands, such as for saving files or quitting the editor.

latency time: The time taken by a disk to spin in order to bring the right sector under the read/write head is called the latency time for the disk. It is dictated by the rotation speed of the disk.

lazy locking: In-version control systems that allow multiple users to check out a file for editing. lazy locking does not lock the file until the file contents are changed by a user.

legacy code: Program written long ago that has no written documentation describing the purpose of various parts of the program.

librarian: A nickname commonly used for the UNIX `ar` utility that allows you to archive your object files into a single library file and manipulate the archive file in various ways.

library: See **language libraries**.

lightweight process: A kernel visible thread in a UNIX process on Solaris.

line display e-mail system: E-mail systems that allow you to edit one line at a time when you are composing an e-mail message. The UNIX mail utility is a prime example of such a system.

link: A way to connect a file (or directory) to a directory so that the file can be accessed as a child of the directory. The actual file may be in another directory.

link file: A file in UNIX that contains the pathname for a file (or directory). A link file, therefore, "points to" another file. The type of such a file is link (denoted by `l` in the output of the `ls -l` command) (see **symbolic link**).

literal constants: Constant values such as digits, letters, and strings. For example: 103, "A", "x", and "Hello".

little endian byte order: A storage (or transmission) order in which the low byte of a multiple-byte data item (char, int, long, etc.) is stored in the low byte of memory and the high byte of the data is stored in the high byte of memory.

loader program: An operating system program that reads an application from the disk, loads it into the main memory, and sets the CPU state so that it knows the location, in the main memory, of the first program instruction in the main memory.

local area network (LAN): Multiple computing devices interconnected form a LAN if the distance between these devices is small, usually less than 1 km.

local client: A client process that runs on the host that you are sitting in front of.

local file system: File system used for organizing files and directories of a single computer system. By using a local file system on a computer system, you can access files and directories on that system only. (A remote file system allows you to access files on the remote computers on a local network.)

local host computer system: The computer system that you are logged on to.

local variable: A variable that is not accessible outside the executing shell script in which it is defined.

login directory: The directory that you are placed in when you log on.

login name: See **username**.

login process: A process created by the getty process that accepts your password, checks for its validity, and allows you to log on by running your login shell process.

login prompt: A character or a character string displayed by an operating system to inform you that you need to enter your login name and password in order to use the system. In a UNIX system, the getty process displays the login prompt.

login shell: The shell process that starts execution when you log on.

loop: A piece of code that is executed repeatedly.

low-level I/O: I/O done via the system call interface.

low-level programming language: A computer programming language that is closer to the language that a CPU speaks, called the CPU's *instruction set*. When written in English-like words called *mnemonics*, this language is called the *assembly language* for the CPU.

lpd: Short for line printer daemon (see **printer daemon**).

LWP: See **lightweight process**.

machine code: See **machine programs**.

machine cycle: A CPU continuously fetches the next program instruction from the main memory, decodes it to verify if the instruction is valid, and then executes it. This process of fetching, decoding, and executing instructions is known as the *CPU cycle*.

machine language: The instruction set of a CPU denoted in the form of 0s and 1s.

machine programs: The programs written in the instruction set of a CPU and expressed in 0s and 1s.

magic number: A number stored in the disk image of an executable file that describes the type of the executable code in the file.

main buffer: Also known as the *editing buffer* or the *work buffer*; the main repository for the body of text that you are trying to create or modify from some previous permanently archived file on disk.

main thread: The thread caused by the execution of the `main()` function in a program.

mainframe: See **mainframe computer**.

mainframe computer: A computer system that has powerful processing and I/O capabilities and allows hundreds of users to use the system simultaneously.

make rules: The rules that are used by the UNIX `make` utility to compile and link various modules of a software product.

master server: The main server process in a concurrent server (see **concurrent server**).

maximum transmission unit: The maximum size of an IP datagram (packet); dependent on the technology used at the data link layer for connection to other hosts.

MBR (master boot record): In simple terms, MBR loads and executes the GRUB boot loader.

menu bar: A collection of menu choices, arranged in either a horizontal or vertical format, that appears on-screen either permanently or when activated using a mouse button.

message body: The message text of an e-mail message.

message header: An important structural part of an e-mail message that usually appears at the top of the message text. It normally contains information such as sender's and receiver's e-mail addresses, subject, date and time the message was sent, attachments, and e-mail addresses of the people who received carbon copies of the mail message.

metacharacters: See **shell metacharacters**.

method: A function which is defined inside a class body. If called as an attribute of an instance of that class, the method will get the instance object as its first argument (which is usually called `self`).

millisecond: One-thousandth (10^{-3}) of a second.

minicomputer: A midrange computer that is more powerful than a PC but less powerful than mainframe computers. Like mainframe computers, the minicomputer also allows multiple users to access the system at the same time.

mirror: A vdev that stores identical copies of data on two or more disks. If any disk in a mirror fails, any other disk in that mirror can provide the same data.

mode control word: A string of characters used with the `chmod` command to specify file privileges.

modeless editor: A text editor, such as emacs, that does not have specific modes for entering text or executing commands that affect the state of the editor. Contrast with an editor with modes, such as vi, vim, or gvim.

MTU: See **maximum transmission unit**.

multimedia internet mail standard (MIME): An e-mail standard that defines various multimedia content types and subtypes for attachments. In particular, digital images, audio clips, and movie files can be transported via e-mail attachments, even on dissimilar e-mail systems, if the systems are MIME-compliant.

multiport router: A router that can interconnect more than two networks.

multiprogramming: In a computer system, the mechanism that allows the execution of multiple processes by multiplexing the CPU. Under multiprogramming, when the process currently using the CPU needs to perform some I/O operation, the CPU is assigned to another process that is ready to execute.

multiservice server: A server that offers multiple services, such as the UNIX superserver, `inetd`.

multithreaded kernel: An operating system kernel that offers multiple concurrent instantiation of kernel services; that is, multithreaded services.

mutable object: An object that can change its value but keep its class identity.

name server: A computer system on an Internet site that helps in mapping a domain name to an IP address, or vice versa. Name servers implement the DNS.

named pipe (FIFO): A file-system-persistent channel on UNIX used for communication between related or unrelated processes on a system.

named pipes: Communication channels that can be used by unrelated UNIX processes on the same computer to communicate with each other. The UNIX system call `mkfifo` (`mknod` in older systems) is used to create a named pipe.

nanosecond: One billionth (10^{-9}) of a second.

net mask: A bit mask used by the TCP/IP protocol to identify whether a host is on a remote network or on a local subnet.

network byte order: Same as **big endian byte order**.

network file system (NFS): Client–server software, commonly used on networked UNIX machines, that allows you to access your files and directories from any computer transparently.

network interface card: A circuit board in a computer system that has a link-level protocol implemented in it. For example, a network card with the Ethernet protocol implemented in it (also referred to as the Ethernet card).

network protocol: See **protocol**.

NIC: See **network interface card**.

nice value: A positive integer value used in calculating the priority number of a UNIX process. The greater the nice value for a process, the higher its priority number, resulting in a lower priority.

noclobber option: A feature in the C and Bash shells that forces the shell to ask you to have the shell prompt you before deleting a file when you execute the `rm` command.

nonterminating loop: A loop that does not have a proper termination condition and, therefore, does not terminate. This is usually caused by bad programming, but there are certain applications, such as Internet servers (e.g., Web servers), that must use infinite loops to offer the intended service.

null command: The Bourne shell command "`:`"; does not do anything except to return the value "`True`". When used in a C shell script, this command causes the C shell to execute the remaining script under the Bourne shell.

null string: A string that contains no value. When displayed on the screen, it results in a blank line.

object code: A program generated by the assembler program. It is in the machine language of the CPU in the computer, but the library calls have not yet been resolved. The task of resolving library calls is performed by another program called the *linker* (or *linkage editor*).

open software system: Software whose source code is freely available to the community of users so they can modify it as they wish. An example of such a system is the LINUX operating system.

others: In UNIX, when we talk about a user's access permissions for a file, "others" refers to everyone except the owner of the file and the users in the owner's group.

package management system: A program that installs new or improved applications or utilities on your UNIX system, usually by compiling and linking program modules from packages. An example is App Café on PC-BSD.

package upgrades: Using the `pkg` command on both PC-BSD and Solaris to maintain the user application packages, apps, and programs at their latest release, or delete them, or add new ones.

packages: A collection of program components for an application that can be installed on your UNIX system via the use of a package management system.

packet: A term used for a fixed-size message (containing data and control information) in networking terminology. A TCP packet is called a *segment*, and a UDP or IP packet is called a *datagram*.

panel, the: In the K Desktop management system, a menu bar, found by default at the bottom of the screen display, which contains a set of buttons that accomplish common tasks on the K Desktop.

parallel execution: Simultaneous execution of multiple commands with the help of CPU scheduling. The processes corresponding to all the commands in the command line are executed in the background.

parent process: A process that creates one or more child processes.

passive socket: A socket that listens for incoming connection requests from client processes. It is a socket on which the `listen()` system call has been executed.

password: A sequence of characters (letters, digits, punctuation marks, etc.) that every user of a time-sharing computer system must have in order for him/her to use the system (see **username**).

path: A slash-delimited pathname for the dataset component.

pathname: The specification of the location of a file (or directory) in a system with a hierarchical file system.

PCB: See **process control block**.

per-process file descriptor table (PPFDT): A kernel table that contains an entry each for all open files in a process, including standard files (standard input, standard output, and standard error). It is indexed by a file descriptor. An entry in this table contains a pointer to an entry in the system-wise file table (SFT) for the opened file.

personal computer (PC): A computer system that, typically, allows a single user to use the system at any one time, although some of the newer PCs allow multiple users to use the system simultaneously. Examples of such systems are Macintosh and home computers running under DOS, Windows 9X, and LINUX.

PF: A kernel-level software system that screens network packets by checking the properties of individual packets and the network connections built from those packets against the filtering rules defined in its rule configuration files. The packet filter arbitrates the disposition of those packets. This could mean passing them through or rejecting them, or it could trigger events that parts of the operating system or external applications work on to dispose of the packets.

physical communication medium: The medium used to connect the hardware resources (computers, printers, etc.) on a network. It includes telephone lines, coaxial cable, glass fiber, a microwave link, and a satellite link.

pipe: A process-persistent channel on UNIX used for communication between related processes on a system.

pipe character (|): The symbol used to connect the standard output of a command to the standard input of another command in a shell script or while using a shell interactively.

point-and-click device: Under a graphical user interface, a device is needed to point to an icon, button, window, or any other part of a window and press (click) a button on the device to perform an operation such as executing a program. Joysticks and mouses are examples of point-and-click devices.

pool: Identifies the name of the storage pool that contains the dataset.

port number: A 16-bit integer number associated with every Internet service, such as TELNET. Port numbers are maintained by TCP and UDP. Well-known services such as FTP, HTTP, and TELNET have well-known ports associated with them. The port numbers for some well-known services are: 21 for FTP, 80 for HTTP, and 23 for TELNET.

port number: A port number is a positive integer in the range 0 to 65535 used to distinguish different services offered on a host.

portability: The ability to move the source code (see **source code**) for a system easily and without major modifications from one hardware platform to another.

positional parameters: Shell environment variables $1–$9 that can be used to refer to the command line arguments with which a shell script is executed.

POSIX (Portable Operating System Interface): A family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems. POSIX defines the API, along with command line shells and utility interfaces, for software compatibility with variants of UNIX and other operating systems.

POST: After the power to an x86 system is turned on, the firmware executes a power-on self-test (POST), locates and installs firmware extensions from peripheral board ROMS, and then begins the boot process through a firmware-specific mechanism.

post office protocol (POP): A method of accessing e-mail messages at a mail server by using client software to download the messages to the client machine for reading offline.

PPFDT: See **per-process file descriptor table**.

present working directory: Also known as the **current directory**; the directory that you are in at a given time. You can use the `pwd` command to display the full pathname of this directory.

print queue: A queue associated with every printer where incoming print requests are queued if the printer is busy printing, and printed one by one as the printer becomes available.

printer daemon: See **printer spooler**.

printer spooler: A system process running in the background that receives print requests and sends them to the appropriate printer for printing. If the printer is busy, its request is put in the printer's print queue.

proc: The structure part of a UNIX process's PCB that contains the scheduling-related information for the process. It always remains in memory regardless of the state of the process.

process: An executing program.

process address space: The main memory space allocated to a process for its execution. When a process tries to access (read or write) any location outside its address space, the operating system takes over the control, terminates the process, and displays an error message that informs the user of the problem.

process control block: A kernel data structure that keeps track of the runtime attributes of the process.

process-persistent object: An object that remains in existence for the life of the process that created it.

processor scheduler: A piece of code in an operating system that implements a CPU scheduling algorithm.

program control flow commands: See **program control flow statements**.

program control flow statements: The shell commands (statements) that allow the control of a shell script to go from one place in the program to another. Examples of such statements are `if-then-else-fi` and `case`.

program generation tools: Software tools and utilities that can be used by application programmers to generate program and executable files. Examples of such tools are editors and compilers.

program header: Important notes at the top of a program file that include information like file name, date program was written and last modified, author's name, purpose of program, and a very brief description of the main algorithm used in the program.

protocol: A set of rules used by computers—network protocols in the operating system software or network applications—to communicate with each other. Some of the commonly used protocols in the networking world are ATM, Ethernet, FTP, HTTP, IP, SMTP, TCP, TELNET, and UDP.

protocol port number: See **port number**.

pseudodevices: Special devices in the **/dev** directory that simulate physical devices.

pseudonym: Also known as an **alias**, a nickname given to a command or e-mail address.

public-key cryptography: An encryption technique that uses two keys: A public key and a private key. The private key is kept on your computer and is used to decode encrypted messages. The public key is made available to anyone who wants to decrypt your messages.

python iterable: An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as a list, string, and tuple) and some non-sequence types such as `dict` and `file` and objects of any classes you define with an `_ _iter_ _()` or `_ _getitem_ _()` method.

pythonic: An idea or piece of code which closely follows the most common usages of the Python language, rather than implementing code using structures common to other languages.

quantum: See **time slice**.

queue: An arrangement of items/requests/messages for serving them on the first-come-first-served (FCFS) basis.

race condition: A condition under which threads in concurrent cooperating processes do not access data mutually exclusively; the final result produced by a process is dependent on the order in which instructions in different threads access the shared data, and the results may or may not be correct.

RAIDZ: A virtual device that stores data and parity on multiple disks.

random access memory (RAM): A storage place inside a computer system that is divided into fixed-size locations where each location is identified by a unique integer address and any location can be accessed by specifying its address. Although there are RAMs in various I/O devices, RAM is normally used for the main memory in a computer system, which is also a read–write memory.

read permission: The read permission on a UNIX file allows a user to read the file. The read permission on a directory allows us to read the names of files and directories in the directory.

real-time computer system: A computer system that must generate output for a command within a specified interval of time, else the output is useless.

redirection operator: An operator used in a UNIX shell for attaching the standard input, standard output, and standard error of a process to a desired file (see **standard files**).

registers: Temporary storage locations inside a CPU that are used by it as scratch pads.

regular expression: A set of rules that can be used to specify one or more items in a single character string (sequence of characters). Many UNIX tools such as awk, egrep, fgrep, grep, sed, and vi support regular expressions.

relocatable code: An executable code that would run regardless of where it is loaded into the memory.

remote client: A client process running on a host connected to your server via a network connection.

requests for comments (RFCs): Technical documents describing the Internet architecture, TCP/IP protocol suite, new protocols, revised protocols, and other Internet-related information items. Initial versions of RFCs are called *Internet drafts*.

resilvering: The process of copying data from one device to another device. For example, if a mirror device is replaced or taken offline, the data from an up-to-date mirror device is copied to the newly restored mirror device. This process is referred to as mirror resynchronization in traditional volume management.

resource manager: The operating system is also known as the resource manager because it allocates and deallocates the computer resources in an efficient, fair, orderly, and secure manner.

resource utilization: The utilization of a resource (usually a hardware resource, such as the CPU) is the percentage of the time it has been in use since the computer system has been running.

rlogin: A UNIX network protocol that allows you to log on to another host on a local area network.

root: The login name of the superuser (see **superuser**) in a UNIX system.

root directory: The directory under which hang all the files and directories in a computer system with a hierarchical file system. Thus, it is the “grandparent” of all the files and directories.

root window: The window under which all other windows are opened as its children.

round-robin scheduling algorithm: A CPU scheduling algorithm in which a process gets to use the CPU for one quantum and then the CPU is given to another process.

This algorithm is commonly used in time-sharing systems like UNIX and LINUX to schedule multiple processes on a single CPU.

route: The sequence of routers that a packet goes through before it reaches its destination.

router: A special host on an internet that interconnects two or more networks and performs routing of packets (called *datagrams* in the TCP/IP terminology) from the sender host to the receiver host. Routers are also called *gateways*.

rsh: A UNIX network protocol that allows you to execute a command on another computer on a local area network.

runtime performance: The time and space taken by a program to finish its execution.

search path: A list of directories that your shell searches to find the location of the executable file (binary or shell script) to be executed when you type an external command at the shell prompt and hit the `<Enter>` key.

sector: Disks are read and written in terms of blocks of data, known as sectors. The typical sector size is 512 bytes.

seek time: The time taken by the read/write disk head to move laterally to the desired track (cylinder) before a read or write operation can take place.

sequential execution: One-by-one execution of commands; one command finishes its execution and only then does the execution of the second command start.

server software: In the client–server software model, the server process computes the response for a client request and sends it to the client, who handles it according to the semantics of the command. All Internet services are implemented on the basis of a client–server software model. An example of a server software is a Web server.

session leader: The login shell process.

set-group-ID (SGID) bit: A special file protection bit which, when set for an executable file, allows you to execute the file on the behalf of the file's group. Thus, you execute the file with group privileges.

set-user-ID (SUID) bit: A special file protection bit which, when set for an executable file, allows you to execute the file on the behalf of the file's owner. Thus, you execute the file with the owner's privileges.

SFT: See **file table** and **system-wide file table**.

SGID: See **set-group-ID**.

sh: The abbreviation of Bourne shell.

shell: A computer program that starts execution when the computer system is turned on or a user logs on. Its purpose is to capture user commands (via the keyboard under a CUI and via a point-and-click device under a GUI) and execute them.

shell environment variables: Shell variables used to customize the environment in which your shell runs and for proper execution of shell commands.

shell metacharacters: Most of the characters other than letters and digits have special meaning to a shell and are known as shell metacharacters. They are treated specially, and therefore cannot be used in shell commands as literal characters without specifying them in a particular way.

shell prompt: A character or character string displayed by a shell process to inform you that it is ready to accept your command. The default shell prompt for Bourne shell is `$` and for C it is `%`. You can change a shell prompt to any character or character string.

shell script: A program consisting of shell commands.

shell variable: A memory location that is given a name which can then be used to read or write the memory location.

signal: In UNIX jargon, a mechanism that allows interruption of a process. It is also known as **software interrupt** in computer science literature.

Simple Mail Transfer Protocol: See **SMTP**.

single stepping: A feature in symbolic debuggers that allows you to stop program execution after every instruction execution. The next instruction is executed by using a command. This is also sometimes called *tracing program execution*.

single UNIX specification (SUS): A family of standards for computer operating systems, compliance with which is required to qualify for the name "UNIX." The core specifications of the SUS are developed and maintained by the Austin Group, which is a joint working group of IEEE, ISO JTC 1 SC22, and The Open Group.

single-threaded process: A process that only has the main thread.

slave process: A child process created by the master process to handle a client request. See **master server**.

slice: A disk partition created with partitioning software.

SMTP: Stands for Simple Mail Transfer Protocol, which is the protocol used in all e-mail systems (e.g., elm, mail, and pine) running on the Internet.

snapshot: An optional component that identifies a snapshot, or exact duplicate, of a dataset.

sniffing: Also known as *packet sniffing*; the equivalent of wiretapping a telephone conversation for Internet traffic.

socket: A process-persistent endpoint of communication on UNIX used for communication between related or unrelated processes on a system or on different systems on a network.

socket address: The address of a UNIX domain socket is a pathname and that of an Internet domain socket is the IP address of the host on which the socket resides plus a protocol port number.

socket descriptor: A descriptor for a socket that is used as an index into the per-process file descriptor table. See **per-process file descriptor table**.

socket name: See **socket address**.

soft link: See **symbolic link**.

software cost model: A model used to estimate the cost of a software product.

software interrupt: A mechanism used in UNIX to inform a process of some event, such as the user pressing `<Ctrl-C>` or logging out.

software life cycle: A sequence of phases used to develop a software product. These phases normally consist of analysis of the problem, specification of the product, design of the product, coding of the product, testing of the software, installation of the product, and maintenance of the product.

sort key: A field, or a portion of an item, used to arrange items in sorted order (see **sorting**). For example, the social security number can be used as the sort key for sorting employee records in an organization.

sorting: Arranging a set of items in ascending or descending order by using some sort key.

source code: A computer program written in a programming language to implement the solution for a problem.

source code control system: A UNIX tool for version control.

special character: A character that when used in a command is not treated literally by the command. An example of such a character is `c` in the System V–compliant echo command that forces the command to keep the cursor on the same line.

special file: The UNIX files that correspond to devices (see **block special files** and **character special files**). These files are located in the **/dev** directory.

spoofing: Creating the TCP/IP packets using some other machine's IP address. This is also known as *IP spoofing*. The term *Web spoofing* is used to describe a situation where an attacker creates a shadow "copy" of the entire World Wide Web.

SSD: Stands for Solid State Drive, which is the modern replacement for spinning hard disks.

SSH: A Secure Shell protocol used to send/receive information using highly secure encryption between local or remote users.

standard error: See **standard files**.

standard files: The files where the input of a process comes from and its output and error messages go to. The standard file where a process reads its input is called *standard input*. The process output goes to *standard output*, and the error messages generated by a process go to *standard error*. By default, the standard input comes from your keyboard, and standard output and standard error are sent to the display screen.

standard input: See **standard files**.

standard output: See **standard files**.

start-up file: A file that is executed when you log on or when you start a new shell process. These files belong to a class of files, called **dot** or **hidden files**, as their names start with a dot (`.`) and they are not listed when you list the contents of a directory by using the `ls` command. Some commonly used start-up files are **.bashrc** (start-up file for Bash), **.cshrc** (start-up file for C shell), **.profile** (executed when you log on to a System V–compliant UNIX system), and **.login** (executed when you log on to a BSD-compliant system). All of these files reside in your home directory.

states: The states (conditions) a process can be in, such as running, waiting, ready, and swapped.

static analysis: Analysis of a program that involves analyzing its structure and properties without executing it.

static linking: Linking carried out at compile time.

sticky bit: When an executable file with the sticky bit on is executed, the UNIX kernel keeps it in the memory for as long as it can so that the time taken to load it from the disk can be saved when the file is executed the next time. When such a file has to be taken out of the main memory, it is saved on the swap space (see **swap space**), thus resulting in less time to load it into the memory again.

strong cryptography: An encryption method that cannot be penetrated by anyone except those who have the decryption key.

subnet mask: See **net mask**.

subshell: A child shell executed under another shell.

sudo: A command that allows a permitted user to execute a command as the superuser, or to assume the role of another user, as specified by security policy in a special file.

SUID: See **set-user-ID**.

supercomputer: The name used for most powerful computers that typically have many CPUs and are used to solve scientific problems that would take a long time to complete on smaller computers. Supercomputers are used in organizations such as NASA and various U.S. national laboratories.

superuser: A special user in every UNIX system who can access any file (or directory) on the system. This user is the system administrator, commonly known as the superuser on the system.

sure kill: Sending signal number "9" to a process. This signal cannot be intercepted by the process receiving the signal and the process is definitely terminated.

swap space: An area set aside on the disk at system boot time where processes can be saved temporarily in order to be reloaded into the memory at a later time. The activity of saving processes on the swap space is called *swap out*, and of bringing them back into the main memory is known as *swap in*. The time taken to load a process from the swap space into main memory is less than the time taken to load a file from the disk when it is stored in the normal fashion.

swapper process: A process that swaps in a process from the swap space into the main memory, or swaps out a process from the main memory to the swap space. See **swap space**.

symbolic constant: A constant value that is given to a name so that the name can be used to refer to the value.

symbolic debugger: A software tool that allows you to debug your program as the program runs.

symbolic link: When a symbolic link to a shared file (or directory) is created in a directory, a link file is created that contains the pathname of the shared file. The link file, therefore, "points to" the shared file. The `ln -s` command is used to create a symbolic link.

synchronous I/O: The I/O based on blocking calls and signals.

syslogd: Logging and log files refer to the recording of general and specific actions and events on a UNIX system. A system program generates a call to write to a specific log file, either locally or across the network. The syslogd daemon handles that write call and is guided by entries in **/etc/syslog.conf**.

system administration: For UNIX systems, generally composed of installing the system and configuring it, arranging boot management and maintenance, maintaining the user base, adding postinstall hardware, backing up system and user files, updating the operating system, and upgrading and maintaining installed software, monitoring and tuning system performance, and making the system secure on a network.

system bus: A set of parallel wires used to take bits from the CPU to a device, or vice versa.

system call: A mechanism that allows a process to perform such privileged tasks that it is not allowed to perform by directly accessing (reading or writing) an I/O device or executing a piece of kernel code. A system call is an entry point into the operating system kernel code. System calls can be used by application programmers to have the kernel perform the tasks that need access to a hardware resource, such as reading a file on a hard disk.

system mailbox file: A file that contains all of the e-mail messages that the system has received for you. It is usually under the **/usr/spool/mail** directory, in a file with your login name.

system programming: The ability of writing the kernel code to manage system hardware including main memory and disk space management, disk formatting and defragmentation, CPU scheduling, and management of I/O devices through device drivers.

system updates: Maintaining the operating system itself so that it is at the latest available stable release.

system-wide file table (SFT): See **file table**.

TCP/IP: See **Transport Control Protocol/Internet Protocol**.

TCP/IP protocol suite: See **Transport Control Protocol/Internet Protocol**.

TELNET: An application-level Internet protocol that allows you to log on to a remote host on the Internet.

text editor: Allows you to view and edit (add or delete text in) text files. In spite of all the fuss created by word processors and desktop publishing systems, text files remain the most critical part of computing. They are needed to store source programs written in any type of language (e.g., C, C++, Java, Assembly, and Perl), e-mail messages, test data, and program outputs.

text-driven operating system: An operating system that takes commands to be executed from the keyboard.

theme: In a window manager, the style and appearance of windows and their accompanying components.

thread: A flow of control in an executing program.

thread-safe function: A function that can be called simultaneously from multiple threads, even when the invocations use shared data, because all references to the shared data are serialized.

threshold priority: A positive integer number used in the expression for calculating the priority number of a process in the UNIX scheduler. It is the smallest priority number for a user-level process. All system processes have priority numbers less than the threshold priority.

throughput: The number of processes finished in a computer system in unit time.

tiled display: A technique to arrange windows on the display screen so that they are opened next to each other, just like the tiles on a floor.

time slice: In a time-sharing system, a time slice, also known as a *quantum*, is the amount of time a process uses the CPU before it is given to another process.

time-sharing system: A multiuser, multiprocess, and interactive operating system. UNIX and LINUX are the prime examples of time-sharing systems.

TLD: See **top-level domain**.

top-level domain: The rightmost string in a domain name.

topology: The physical arrangement of hosts in a network. Some commonly used topologies are bus, ring, mesh, and general graph.

transparent encryption: Encryption/decryption automatically done by Secure Shell (SSH) without the knowledge of the sender and receiver processes.

Transport Control Protocol/Internet Protocol: The suite of communication and routing protocols that are the basis of the Internet. They include many protocols such as FTP, ICMP, IP, TCP, TELNET, and UDP.

transport layer interface (TLI): The equivalent of BSD sockets in System V–compliant UNIX systems, it allows processes on the Internet to communicate with each other.

trap: The CPU generated signal to handle an exception in the code being executed. See **internal signal**.

trusted host: Some remote login protocols allow login from a set of hosts without verifying passwords. Such hosts are known as trusted hosts.

u area: Part of a UNIX process's PCB that contains information about signal handling, resources allocation, and a reference to the proc structure.

UEFI (unified extensible firmware interface): A specification that defines a software interface between an operating system and platform firmware. UEFI replaces the basic input/output system (BIOS) firmware interface.

universal resource locator: Protocol://IP_address/pathname or protocol://FQDN/pathname. The protocol field usually contains "http", but can contain "ftp" or "telnet". The pathname field is used to identify the location of a file on the host. URLs are commonly used to identify the location of a Web page to be displayed on your screen. An example of a URL is http://www.up.edu/index.html. In this example, the protocol is http, the FQDN is www.up.edu, and the pathname is ~/index.html.

UNIX domain socket: A socket (an interprocess communication endpoint in BSD-compliant UNIX systems) that can be used for communication between processes on the same computer system.

upstream branch: The default branch that is merged into the branch in question (or that the branch in question is rebased onto). In terms of data, your repo is "downstream" of data coming from upstream repos that you "pull from" and going back to upstream repos that you "push to."

URL: See **universal resource locator**.

user: In UNIX jargon, the term used for the owner of a file when we talk about file access privileges.

user-defined macros: In context with the `make` utility, a user-defined macro is usually a collection of files, keystrokes, compiler options, or compiler names that is given a name. This allows users to access these macros by using `$(macro _ name)` syntax. This capability enhances the readability of the **makefile** and allows you to use the named items at any time—as many times as you want.

user-defined variables: These shell variables are used within shell scripts as temporary storage places whose values can be changed when the script executes.

userid: Every user in a time-sharing system, such as UNIX, is assigned an integer number called his/her userid.

user-level thread: A thread created by a user program using these libraries that is not known to the kernel and is managed solely by the user-level threads libraries.

username: A name by which a user of a multiuser computer system is known to it. Before you can use the computer system, you must enter your username at the login prompt and hit the `<Enter>` key, followed by entering your password and hitting the `<Enter>` key.

vdev (virtual device): A whole disk, a disk partition, a file, or a collection of any of these, usually all of the same type. On PC-BSD, there is no performance disadvantage for using disk partitions rather than entire disks. On Solaris, the write cache is disabled for partitions, thus incurring a performance penalty. In both systems, using files as vdevs is discouraged, except for testing purposes. A collection of vdevs in standardized configurations is known as a **mirror**.

version control: In general, the task of managing revisions to any soft product such as documentation for a product; but, in particular, the task of managing revisions to a software product.

virtual connection: Said to be established between client and server processes when the two have made an initial contact to exchange each other's location (usually IP address and protocol port number). The connection request is almost always initiated by the client process. After the virtual connection has been established, the server process understands that it will receive service requests(s) from the client process. The virtual connection is broken when the client process has received the response to its last request and initiates a request for closing the virtual connection.

virtual machine (VM): An instance of a complete, encapsulated operating system, including a kernel, that can run as a guest under another operating system.

virtual machine guest: The operating system that is accommodated by a host.

virtual machine host: The operating system that accommodates a guest operating system in a virtual environment.

virtual machine monitor (VMM): The operating system supervisory or management software that mediates between virtual machine host and guests. Sometimes called a *hypervisor.*

volume: A dataset that represents a block device. For example, you can create a ZFS volume as a swap device.

Web browser: An Internet application that allows you to surf the Web by allowing users to view Web pages (among other things).

well-known application: An application that is built around a communication protocol described in a request for comments (RFC) such as Hyper Text Transfer Protocol (also called HTTP and WWW), File Transfer Protocol (FTP), Simple Mail Transfer Protocol (SMTP), and Secure Shell (SSH).

well-known port: A port on which a well-known service runs.

well-known service: A service that runs on a well-known port. See **well-known application**.

wide area network (WAN): Also known as a *long-haul network*; a network that connects computing resources that are thousands of kilometers apart, typically spanning several states, countries, or continents.

widowed pipe: A pipe that has one end closed. See **pipe**.

window manager: The graphical management interface, which controls the display of and organizes all client windows on the X server.

window system: A graphical system that provides the generic features of a GUI.

write permission: On a UNIX file, allows a user to write to the file, thus allowing the insertion or deletion of its contents and its removal from the system. The write permission on a directory allows us to create a new file (or directory) under it.

X Window System: A graphical intermediary between you and the UNIX operating system. It was developed at MIT in 1983 as part of the Athena project. It is the de facto GUI for UNIX systems that comes as part of the operating system package.

X Window System event/request model: In the X Window System, an X server is the hardware and/or software that actually takes input from and displays output to the user. The X client is an application program that connects to the server, and receives input events from the server and makes output requests to the server.

yank: Marking/saving one or more lines of text in a file under the vi editor to be pasted elsewhere in the file.

Zettabyte file system (ZFS): See **ZFS file system**.

ZFS clone: A file system whose initial contents are identical to the contents of a snapshot.

ZFS file system: A ZFS dataset of type filesystem that is mounted within the standard system namespace and behaves like other file systems.

ZFS pool: A logical group of devices describing the layout and physical characteristics of the available storage. Diskspace for datasets is allocated from a pool.

ZFS snapshot: A read-only copy of a file system or volume at a given point in time.

zombie: See **zombie process**.

zombie process: A UNIX process that has terminated but still has some system resources allocated to it. Thus, a zombie process results in wastage of system resources. It is usually created when its parent process terminates before it finishes execution.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Index